Objektorientierte Softwareentwicklung

Sven Eric Panitz
Hochschule RheinMain

Version 813

Generiert von LectureNotes Teaching System
9. Oktober 2015

Diese Skriptversion entsteht für das WS2013/14 neu. Es basiert aber weiterhin in Teilen auf den Skripten, die zwischen 2002 und 2012 für unterschiedliche Vorlesungen im Bereich der Softwareentwicklung entstanden sind.

Inhaltsverzeichnis

1 Einführung in die Welt der Softwareentwicklung 7
1.1 Aspekte der Softwareentwicklung . . . 8
1.2 Programmiersprachen . . . 14
1.3 Arbeiten mit der Kommandozeile . . . 16
1.3.1 Basisbefehle . . . 17
1.3.2 Nützliche Standardprogramme . . . 25
1.3.3 Erste Java Programme auf der Kommandozeile . . . 29

2 Grundkonzepte der Objektorientierung 35
2.1 Objektorientierte Modellierung . . . 35
2.2 Klassen und Objekte . . . 37
2.2.1 Felder . . . 39
2.2.2 Objekte . . . 40
2.2.3 Objekte der Klasse `String` . . . 43
2.2.4 Methoden . . . 44

3 Imperative und funktionale Konzepte 47
3.1 Primitive Typen . . . 47
3.1.1 Zahlenmengen in der Mathematik . . . 48
3.1.2 Zahlenmengen im Rechner . . . 48
3.2 Ausdrücke . . . 51
3.2.1 Arithmetische Operatoren . . . 54
3.2.2 Vergleichsoperatoren . . . 55
3.2.3 Logische Operatoren . . . 55
3.2.4 Der Bedingungsoperator . . . 56
3.3 Anweisungen . . . 57
3.3.1 Fallunterscheidungen . . . 57
3.3.2 Iteration . . . 61
3.4 Rekursion . . . 68
3.4.1 Rekursion und Schleifen . . . 70
3.4.2 Einsatz von Rekursion . . . 73

4 Weiterführende Konzepte der Objektorientierung 77
4.1 Vererbung . . . 77
4.1.1 Hinzufügen neuer Eigenschaften . . . 79
4.1.2 Überschreiben bestehender Eigenschaften . . . 79

4.1.3 Konstruktion 80
4.1.4 Zuweisungskompatibilität 81
4.1.5 Späte Bindung (late binding) 83
4.2 Pakete 91
4.2.1 Paketdeklaration 91
4.2.2 Übersetzen von Paketen 92
4.2.3 Starten von Klassen in Paketen 93
4.2.4 Das Java Standardpaket 94
4.2.5 Benutzung von Klassen in anderen Paketen 94
4.2.6 Importieren von Paketen und Klassen 94
4.2.7 Statische Imports 96
4.2.8 Sichtbarkeitsattribute 97
4.3 Schnittstellen (Interfaces) und abstrakte Klassen 102
4.3.1 Schnittstellen 102

5 Graphische Benutzeroberflächen mit Swing 109
5.1 Swings GUI-Komponenten 110
5.1.1 Top-Level Komponenten 111
5.1.2 Zwischenkomponenten 111
5.1.3 Atomare Komponenten 112
5.2 Gruppierungen 113
5.2.1 Flow Layout 114
5.2.2 Border Layout 115
5.2.3 Grid Layout 115
5.3 Eigene GUI-Komponenten 117
5.3.1 Fraktale 119
5.4 Reaktion auf Ereignisse 123
5.4.1 Der `ActionListener` 124
5.4.2 Innere und Anonyme Klassen 125
5.4.3 Lambda Ausdrücke 127
5.4.4 Mausereignisse 128
5.4.5 Fensterereignisse 130
5.5 Zeitgesteuerte Ereignisse 131
5.5.1 Animationen 134
5.6 Weitere Swing Komponenten 139

6 Weiterführende Konzepte 143
6.1 Reihungen (Arrays) 143
6.1.1 Deklaration von Reihungen 143
6.1.2 Erzeugen von Reihungen 144
6.1.3 Zugriff auf Elemente 144
6.1.4 Ändern von Elementen 145
6.1.5 Die For-Each Schleife 145
6.2 Generische Typen 146

6.2.1 Generische Klassen 146
6.2.2 Vererbung 149
6.2.3 Beispiel einer eigenen Listenklasse 152
6.2.4 Standard Sammlungsklassen 154
6.3 Ein- und Ausgabe 157
6.3.1 Dateibasierte Ein-/Ausgabe 158
6.3.2 Textcodierungen 161
6.3.3 Gepufferte Ströme 164
6.3.4 Lesen von einem Webserver 165
6.3.5 Ströme für Objekte 166
6.4 Ausnahmen 168
6.4.1 Ausnahme- und Fehlerklassen 168
6.4.2 Werfen von Ausnahmen 168

7 Zusammenfassung und Ausblick 173
7.1 Fragen und Antworten 173

Kapitel 1

Einführung in die Welt der Softwareentwicklung

Sie haben sich für das Studium eines Informatikstudienganges entschieden. Sie werden damit u.a. in den verschiedenen Modulen verschiedene Aspekte zur Entwicklung von Software kennenlernen. Dieses wird eines der zentralen Inhalte des Studiums sein. Und in diesem Modul werden die Grundlagen hierfür gelegt.

Aufgabe 0 Bilden Sie Paare in der Praktikumsstunde. Interviewen Sie sich gegenseitig paarweise. Anschließend stellen Sie ihren Interviewpartner der ganzen Praktikumsgruppe vor. Bauen Sie bei der Vorstellung eine Lüge über die vorgestellte Person ein. Lassen Sie das Plenum raten, welches die Lüge war.

Aufgabe 1 Chuck Norris wird Vieles nachgesagt. Unter anderem auch, dass er schon einmal bis unendlich gezählt hat. Wenn er tatsächlich in der Lage ist, bis unendlich zu zählen, so wird er das sicher auch zweifach können, indem er einfach zählt:

$$1, 1, 2, 2, 3, 3, 4, 4, 5, 5, \ldots$$

Entsprechend kann er auch dreifach, vierfach, fünfach etc. bis Unendlich zählen. Überlegen Sie eine Zählweise, mit der Chuck Norris dann auch unendlich oft bis Unendlich zählen kann.

Aufgabe 2 Sie haben ein Navigationssystem für die Schifffahrt entwickelt. So ganz zufrieden sind Sie mit Ihrem System aber noch nicht, denn es schlägt Ihnen eines Tages folgende Route vor:

Segeln Sie erst 100km nach Norden, dann 100km nach Westen, anschließend 100km nach Süden und dann 100km nach Osten. Dann haben Sie Ihr Ziel erreicht.

Können Sie sich den Weg sparen?

Aufgabe 3 Sie nehmen bei einer Quizshow Teil. Sie sind der letzte Kandidat. Es gibt zwei Türen. Hinter der einen Tür steht für Sie ein Eimer voll Gold, hinter der anderen Tür lauert ein gefährliches Monster, das Sie fressen will. Vor den Türen steht jeweils ein Wächter. Einer der beiden Wächter sagt immer die Wahrheit, der andere lügt bei allem was er sagt. Sie wissen aber nicht, wer

von den beiden der Lügner ist. Sie dürfen einem Wächter genau eine Frage stellen, die mit Ja oder Nein zu beantworten ist. Was für eine Frage würden Sie stellen, um heraus zu bekommen, hinter welcher Tür das Gold ist.

Aufgabe 4 Sie fahren in ein fremdes Land und erlernen die Sprache. Das Land hat aber teilweise für Sie ungewohnte Begriffe. So gibt es die beiden Begriffe *grot* und *rün*, die Farben bezeichnen. Als *grot* werden alle Dinge bezeichnet, die bis zum 1.6.2047 grün sind und dann die Farbe nach rot wechseln. Als *rün* werden alle Dinge bezeichnet, die bis zum 1.6.2047 rot sind und dann die Farbe nach grün wechseln. Das erscheint Ihnen vollkommen unlogisch und irritierend und sie versuchen den Bewohnern des Landes zu erklären, dass diese Begriffe blödsinnig sind. Die Bewohner kontern aber damit, dass die Begriffe *rot* und *grün* vollkommen willkürlich und unlogisch sind. Dabei erklären Sie die Begriffe *rot* und *grün* mit dem Begriffen *grot* und *rün*.

Wie sieht diese Begriffserklärung aus?

Aufgabe 5 Es gibt ein Rechenverfahren für die Multiplikation, die sogenannte vedische Multiplikation, die für bestimmte Produkte besonders einfach ist. Die Idee basiert auf der Tatsache, dass wir mit Zehnerpotenzen besonders einfach rechnen können. Rechenverfahren werden in der Informatik als Algorithmus bezeichnet:

Vedische Multiplikation: Zur Multiplikation zweier Zahlen $a * b$ wähle eine Zehnerpotenz 10^n, so dass die Summe der Beträge von $a - 10^n$ und $b - 10^n$ besonders klein ist. Es sei $\bar{a} = a - 10^n$ und $\bar{b} = b - 10^n$.

Für das Produkt rechne: $a * b = (b + \bar{a}) * 10^n + \bar{a} * \bar{b}$.

Dieser Rechenweg wird besonders einfach, wenn $\bar{a}$ und $\bar{b}$ betragsmäßig kleine Zahlen sind, denn dann ist nur noch eine sehr simple Multiplikation durchzuführen und ansonsten Additionen.

Rechnen Sie folgende Aufgaben mit der vedischen Multiplikation:

a) 999*999

b) 87*101

c) 101*102

1.1 Aspekte der Softwareentwicklung

> Ich schreibe. Ich bin Software-Autor. [...] Ich schreibe ab jetzt nur noch Software, die was mit meinem Leben zu tun hat. Autobiographisch.
>
> *Kristof Magnusson: Männerhort*

Das obige Zitat aus der Komödie »Männerhort« ist näher an der Realität, als man denken mag; denn im Prinzip ist Software nichts weiter als ein Text, den ein oder mehrere Autoren geschrieben haben. Insofern ist es durchaus legitim bei Programmierern von Autoren zu sprechen. Und je mehr sich die Autoren mit ihrer Software identifizieren, desto besser ist meistens auch die Qualität der Software. Dieses gilt oftmals besonders für quelltextoffene Software. Hier steht immerhin der gute Ruf der Autoren auf dem Spiel.

Auf der einen Seite scheint die Entwicklung von Software einfach zu sein. Es ist ein Text zu schreiben. Hierzu wird eigentlich kein anderes Werkzeug als ein Texteditor benötigt. Andererseits ist die Entwicklung von Software ein sehr komplexer Vorgang, denn ein Softwaresystem ist oft sehr umfangreich und besteht aus vielen Einzelkomponenten. Um dieser Komplexität Herr zu werden, hat die Informatik über die letzten Jahrzehnte viele Techniken und Prozesse entworfen, die bei der Entwicklung von Software helfen sollen. Im Laufe eines Informatikstudiums werden Sie in verschiedenen Modulen die vielen unterschiedlichen Aspekte der Software-Entwickung kennenlernen. In diesem Modul soll Ihnen hierzu ein Grundüberblick und ein erstes Fundament in der Programmierung gelegt werden.

Die Fragestellungen der Softwareentwicklung reichen von technischen Aspekten, wie die Programmierung von Betriebsmitteln, bis hin zu Prozessen und Rollen beim Arbeiten im Team. Es ist zu klären, für welche Hardware was für Software mit welcher Funktionalität auf welche Weise zu entwickeln ist, wie die Qualität der Software sicher gestellt werden kann, wie die Kosten des Entwicklungsprozesses abgeschätzt werden können und wie die Software zum Kunden gelangt und im laufenden Betrieb gewartet werden kann.

Wir geben eine ungeordnete und sicher nicht vollständige Liste verschiedenster Aspekten der Programmierung:

- **Spezifikation**: Ein großes Thema ist, zunächst einmal festzuhalten, was die zu erstellende Software überhaupt leisten soll. Diesen Schritt der Softwareentwicklung bezeichnet man allgemein als Spezifikation. Spezifiziert wird auf den unterschiedlichsten Abstraktionsebenen. Allgemein für die komplette Software, für einzelne Komponenten oder auch für kleinste Teileinheiten. Es gibt verschiedene formale und semi-formale Sprachen, zur Spezifikation. Im Bachelorstudiengang werden Sie die gängigsten Spezifikationssprachen in dem Modul »Softwaretechnik« kennen lernen. Weitere formale Spezifikationssprachen werden im Masterstudiengang im Modul »Formale Modelle« vorgestellt. Grundlage aller fast Spezifikationssprachen ist ein gewisses formales Handwerkszeug, dass im Modul »Diskrete Strukturen« vermittelt wird.
- **Sicherheit und Datenschutz**: Wer die Tagespresse verfolgt, sieht, dass die Sicherheit von Daten ein großes und ernstes Thema ist. Das Modul »IT-Security« wird hierzu in Ihren Studium die theoretischen Grundlagen dieses Aspektes vermitteln.
- **Ergonomie und Benutzerfreundlichkeit** Jeder Anwender hat sich sicher

schon einmal darüber geärgert, dass Software nicht intuitiv oder folgerichtig zu bedienen war. Das einfache Dinge nur kompliziert und durch viele Schritte realisiert werden konnten oder auch dass die Software die wichtigen Informationen nicht leicht zugänglich und übersichtlich dargestellt hat. Auch eine in der Funktionalität fehlerfreie Software kann unbrauchbar sein, weil sie zu schwer zu bedienen ist. Im Wahlbereich unseres Curriculums findet sich das Modul »Usability Engineering«, dass sich mit diesen Fragen auseinander setzt.

- **Optimierung** Ein wichtiger Aspekt einer Software ist, dass sie nicht nur korrekt funktioniert, sondern auch einen guten Durchsatz und schnelle Antwortzeiten hat. Hierbei kann schnell aber auch ganz unterschiedliches bedeuten. Von eine Handelsplattform für Börsengeschäfte wird eine extrem schnelle Abarbeitung der Anfrage erwartet, ein System das ein dreidimensionalen Film rendered darf durchaus manchmal Stunden brauchen. Egal in welchem Gebiet die Software eingesetzt wird, sie soll möglichst effizient laufen. Hierbei wird es unter Umständen nötig, korrekt laufende Programme zu Optimieren. Optimierungen können den verwendeten Algorithmus betreffen, der in einer hohen Komplexitätsklasse liegt und durch einen effizienteren ersetzt werden kann. Es kann das Zwischenspeichern von Teilergebnissen beinhalten, so dass mehrfaches Errechnen gleicher Zwischenergebnisse entfällt. Es kann aber auch auf einer mikroskopischen Ebene passieren, indem besonders häufig benutze Programmteile schon in kleinsten Teilen auf Assemblerebene optimiert werden. Mit der Komplexität bestimmter Algorithmen beschäftigt sich in unserem Curriculum unter Anderem das Modu »Algorithmen und Datenstrukturen«.
- **Tests** Beim Theater sagt man: »Alles, was nicht geprobt ist, geht schief.« Ähnliches gilt für Software. Alles, was nicht getestet wurde, hat eine hohe Wahrscheinlichkeit, dass es nicht funktioniert. Doch wie testet man Software besten? Reicht es ein paar Anwender das Programm ausprobieren zu lassen. Welche Komponenten lassen sich automatisch reproduzierbar testen? Welche Granularität können die Tests haben? Wird in Kenntnis des Quellcodes getestet oder ohne diese Kenntnis das ein-/Ausgabeverhalten der Software? Und wer testet? Die Entwickler selbst, oder Personen, die die Software nicht entwickelt habe? In diesem Modul werden wir erste Testszenarien für Entwicklertests kennenlernen. Ein umfassender Überblick wird in dem Modul »Softwaretechnik« gegeben.
- **Logging** Besonders für Server-Anwendungen, die über einen langen Zeitraum laufen, ist es wichtig, dass ein Administrator nachverfolgen kann, was über einen bestimmten Zeitraum mit der Anwendung vorgefallen ist. Wie viele Anfragen bedient wurden, ob es Fehlerfälle gab oder auch wie gut der Durchsatz der Anwendung war. Hierzu ist es unerlässlich, dass eine Anwendung ein Logbuch führt, in dem alle wichtigen Schritte des Prorammablaufs dokumentiert sind. Dieses Logbuch ist insbesondere wichtig, wenn es zu Fehlern kam, und die Fehlerursache ermittelt werden muss.

- **Verifikation** Klar, Software soll fehlerfrei laufen. Aber nicht immer genügen Tests, um das fehlerfreie Verhalten eines Programms zu zeigen. Tests können immer nur endlich viele Eingaben eines Programms checken. Ein Programm hat aber potentiell unendlich viele Zustände und Eingaben. Mit formalen Verifikationsmethoden versucht man für besonders kritische Software, die vielleicht eine gefährliche Anlage steuert, mathematisch zu beweisen, dass sie für alle Eingaben immer das gewünschte Verhalten zeigt. Man unterscheidet:
  - *partielle Korrektheit:* wenn das Programm für eine bestimmte Eingabe ein Ergebnis liefert, dann ist dieses bezüglich der Spezifikation korrekt.
  - *totale Korrektheit:* Das Programm ist partiell korrekt und terminiert für jede Eingabe, d.h. liefert immer nach endlich langer Zeit ein Ergebnis.
- **Internationalisierung** Die Welt ist insbesondere durch die Daten-Kommunikationsnetze enger zusammen gerückt. Ein Stück Software wird nicht für ein Land oder einen Kulturkreis allein geschrieben, sondern wird potentiell weltweit eingesetzt. Unter der Internationalisierung (abgekürzt als I18N für internationalization, I 18 Buchstaben N) versteht man, dass die Software in verschiedenen Kulturkreises eingesetzt werden kann. Dieses beinhaltet z.B. dass alle Schriftzeichen in einer Textverarbeitung benutzt werden können oder dass die Schreibrichtung in einem Texteingabefeld eingestellt werden kann, da in einigen Schriften von rechts nach links in anderen von links nach rechts, und teilweise auch von oben nach unten geschrieben wird.
- **Lokalisierung** Die Lokalisierung beschäftigt sich auch mit der globalen Einsetzbarkeit der Software. Hier geht es zusätzlich darum, dass die Benutzerführung in verschiedenen Sprachen vorliegt, zum Beispiel die Menübeschriftungen in mehreren Sprachen vorliegen, die Hilfetexte übersetzt worden. Entsprechend I18N wird Lokalisierung auch als L12N abgekürzt. Kurz gefasst kann man sagen: in einer internationalisierten Software lassen sich auch chinesische Texte schreiben, aber die Knöpfe und Menüs sind weiterhin auf englisch beschriftet. In einer lokalisierten Software gibt es auch Beschriftungen, Hilfetexte etc auf chinesisch.
- **Vorgehensmodell** Wie schreibt man eine Software. Einfach los tippen, bis man fertig ist, wäre ein Vorgehensmodell. Ein Vorgehensmodell legt fest, in welchen Schritten, mit welchen Zwischenergebnissen eine Software im Team erstellt wird. Dabei kann das Team auch aus einer Person bestehen, wie es in der Regel bei einer Bachelorarbeit der Fall ist. Es wurden in der Informatik eine Reihe unterschiedlicher Vorgehensmodelle entwickelt. Auf theoretischer Ebene werden diese Ihnen im Modul »Softwaretechnik« vorgestellt. Praktisch werden Sie spätestens im Wahlprojekt des fünften Semesters nach einem Vorgehensmodell arbeiten.
- **Dokumentation** Schön wäre es, wenn der Quelltext einer Software selbsterklärend wäre. Dieses ist leider selten der Fall, auch wenn es auch eine Bestrebung in der Entwicklung neuer Programmiersprachen ist, dass Programme

verständlicher und selbsterklärender sind. Daher ist es unumgänglich, dass zu einer Software eine Dokumentation existiert. Wobei man drei Adressaten der Dokumentation identifizieren kann:

- **Endanwender**: für die Endanwender wird ein Benutzerhandbuch geschrieben. Hier sind keinerlei Interna der Software beschrieben, sondern lediglich die Funktionalität der Benutzerschnittstelle.
- **Entwickler die Komponenten benutzen**: die meiste Software ist nicht monolithisch, sondern besteht aus einzelnen Komponenten, die in unterschiedlichen Softwareprojekten wiederverwendet werden können. Für diese Komponenten ist die Programmierschnittstelle, das sogenannte API (*application programmers interface*) zu beschreiben. Es ist zu dokumentieren, welche Funktionalität die Komponente anbietet und wie diese von einem Programmierer zu benutzen ist. Welche Vorbedingungen gelten müssen und welche Nachbedingungen nach Aufruf einer Schnittstelle gelten. Interna der Umsetzung sind hierbei nicht dokumentiert.
- **Entwickler der Software selbst:** Sie werden schon in Kürze feststellen, dass Sie Ihre eigenen Programme schon nach wenigen Wochen manchmal gar Tagen nicht mehr verstehen. Sie werden sich frage: Was habe ich mir da nur gedacht? Deshalb ist es sinnvoll, seinen Programmtext zu kommentieren. Innerhalb des Programmtextes sind Kommentarzeilen eingefügt, die erklären, wie ein bestimmter Teil des Codes funktioniert. Diese Kommentare sollen helfen, dass die Kollegen im Team oder man selber, oder Kollegen, die irgendwann in der Zukunft die Software weiterentwickeln, den Programmtext besser verstehen.

In vielen Firmen verlangt wird das API auf Englisch zu dokumentieren und auch Kommentare im Programmtext auf Englisch zu schreiben. Dieses soll erleichtern, dass auch internationale Kollegen mit ins Team aufgenommen werden können oder die Software ins Ausland zur Weiterentwicklung verkauft werden kann. Auch bei einer kleinen Firma, die in Deutschland angesiedelt ist und nur deutsche Mitarbeiter hat, weiß man nie, was die Zukunft bringt. Vielleicht wird die Firma von einer anderen in Korea angesiedelten Firma aufgekauft, vielleicht wird es eine lose Zusammenarbeit mit einer französischen Firma geben. Auf jeden Fall sind solche Schritte weniger verbaut, wenn der Programmtext auf Englisch dokumentiert und kommentiert ist. Es empfiehlt sich also, sich frühzeitig an zu gewöhnen, seinen Quelltext in seinem rudimentären Englisch zu dokumentieren.

- **Wartung** Wenn Sie in ihrem betriebspraktischen Modul in einer Firma sind, die Software entwickelt, werden sie vielleicht feststellen, dass dort mehr Zeit damit verbracht wird, bestehende Software zu Warten als neue Software zu schreiben. Zur Wartung gehört zunächst einmal, dass Fehler, die erst im Betrieb beim Kunden gefunden wurden, korrigiert werden. Es kann sich dabei um echte funktionale Fehlfunktionen, die bis zum Programmabsturz führen

können, handeln aber auch um Probleme der Benutzerfreundlichkeit und der Ausführungsgeschwindigkeit. Einen großen Wartungsaufwand fordern oft neue Versionen des Systems, auf dem die Software installiert ist. Sei es das Betriebssystem, ein Webserver oder auch die Datenbank.

- **Portieren:** Oft wird es nötig, ein Programm auf eine andere Plattform zu portieren. Ein unter Windows erstelltes Programm soll z.B. auch auf Unix-Systemen zur Verfügung stehen.
- **Einsatz eines Debuggers** Bei einem Programm ist immer damit zu rechnen, dass es Fehler enthält. Diese Fehler werden im besten Fall von der Qualitätssicherung entdeckt, im schlechteren Fall treten sie beim Kunden auf. Um Fehler im Programmtext zu finden, gibt es Werkzeuge, die ein schrittweises Ausführen des Programms ermöglichen *(debugger)*. Dabei lassen sich die Werte, die in bestimmten Speicherzellen stehen, auslesen und auf diese Weise der Fehler finden.
- **Architektur** Eine Software besteht aus vielen Einzelkomponenten, die unterschiedliche Teilaufgaben übernehmen und im besten Falle für verschiedene Softwareprojekte wiederverwendbar sind. Die Aufteilung der Software in einzelnen Komponenten wird als Architektur bezeichnet. Es gibt bestimmte Standardarchitekturmuster, die bei ähnlichen Aufgabenstellungen Anwendung finden können. Elementare Programmiermuster werden im Modul »Softwaretechnik« vorgestellt.
- **Kommunikation** Es gibt heute kaum noch Software, die nicht auf irgendeine Art und Weise über ein Netzwerk kommuniziert. Insofern ist ein wichtiger Aspekt der Softwareentwicklung, wie diese Kommunikation gemacht wird. Insbesondere auch wie eine sichere Kommunikation erreicht wird und wie sich die Software bei Ausfall der Kommunikationswege verhalten soll. Die Grundlagen zur Datenkommunikation werden im Modul »Rechnernetze und Telekommunikation« vermittelt. Mit Anwendungen, die stark auf dem Internet basieren, beschäftigt sich das Modul »Webbasierte Anwendungen«.
- **Datenhaltung** Kaum ein Softwareprojekt kommt ohne eine Komponente zur Speicherung umfangreicher Daten aus. man spricht dabei von Persistenz. Daten bleiben erhalten, auch wenn das Programm nicht mehr läuft. Eine einfache Form der Datenpersistenz ist eine Datei, in der die Daten in irgendeiner Form geschrieben werden. In der Regel haben Daten eine Struktur und werden in einer strukturierten Form abgespeichert, um in dieser Struktur effizient suchen zu können. Hierzu gibt Datenbanken. Bereits in zweitem Semester werden Sie entsprechenden Modul die Grundlagen der Datenbanktechnologie zur Software-Entwicklung kennen lernen.
- **Algorithmen** In der Regel erhalten Unterprogramme bestimmte Eingaben, für die eine Ausgabe zu berechnen ist. Der Weg aus den Einzelschritten, wie diese Ausgabe erzeugt wird, wird als Algorithmus bezeichnet. Für viele typische immer wieder benötigte Aufgaben, wie zum Beispiel dem Sortieren ein-

er Liste von Daten, gibt es Standardalgorithmen, auf die in der Software-Entwicklung zurück gegriffen werden kann. Im zweiten Semester im Modul »Algorithmen und Datenstrukturen« werden die elementarsten Standardalgorithmen vermittelt, sowie grundlegende Gedanken über die Komplexität bestimmter Algorithmen.

- **Codierung** Die eigentliche Codierung ist in der Regel der einzige Schritt, der direkt Code in der gewünschten Programmiersprache von Hand erzeugt. Alle anderen Schritte der Programmierung sind mehr oder weniger unabhängig von der zugrunde liegenden Programmiersprache.

  Für die Codierung empfiehlt es sich, Konventionen zu verabreden, wie der Code geschrieben wird, was für Bezeichner benutzt werden, in welcher Weise der Programmtext eingerückt wird. Entwicklungsabteilungen haben zumeist schriftlich verbindlich festgeschriebene Richtlinien für den Programmierstil. Dieses erleichtert, den Code der Kollegen im Projekt schnell zu verstehen.

- **Refaktorierung** Man sagt zwar gerne »Never change a running system.« oder auch »If it works, don not fix it«, doch können wir das in der Software-Entwicklung so nicht stehen lassen. Funktionierende Lösungen können unnötig komplex und unübersichtlich sein, so dass sie auf lange Sicht schwer zu warten sind und auch nicht leicht m weitere Funktionalität erweitert werden kann. Hier ist es oft sinnvoll, den Code zu refaktorieren, d.h. umzuschreiben, so dass er durch eine einfachere und übersichtlichere Lösung ersetzt werden kann. Dieses kann man nur sicher machen, wenn es eine ausreichende Testabdeckung gibt, so dass man zeigen kann, dass die neue Lösung sich ebenso wie die alte Verhält. Oft ist die Refaktorierung das Resultat eines Reviews zum Beispiel in Form eines *code walk-through* bei dem der Entwickler seinen Programmtext zeilenweise zusammen mit anderen Entwicklern durch geht.

Wie Sie sehen, gibt es einiges zu lernen in Ihrem Studium. Auf jeden Fall wird Ihnen nicht langweilig werden. Beginnen wir also mit den ersten Schritten zur Ausbildung zum Software-Autor.

1.2 Programmiersprachen

Auch wenn der letzte Abschnitt gezeigt hat, dass die eigentliche Codierung der Software nur ein kleiner Teil des Projektes ausmacht, sind doch eine oder mehrere Programmiersprachen auszuwählen, in denen das Projekt realisiert wird.

Es gibt mittlerweile mehr Programmiersprachen als natürliche Sprachen.[1] Die meisten Sprachen führen entsprechend nur ein Schattendasein und die Mehrzahl der Programme konzentriert sich auf einige wenige Sprachen. Programmiersprachen lassen sich nach den unterschiedlichsten Kriterien klassifizieren.

[1] Wer Interesse hat, kann im Netz einmal suchen, ob er eine Liste von Programmiersprachen findet.

Im folgenden eine hilfreiche Klassifizierung in fünf verschiedene Hauptklassen.

- **imperativ** (C, Pascal, Fortran, Cobol): das Hauptkonstrukt dieser Sprachen sind Befehle, die den Speicher manipulieren.
- **objektorientiert** (Java, C++, C#, Objective C, Eiffel, Smalltalk): Daten werden in Form von Objekten organisiert. Diese Objekte bündeln mit den Daten auch die auf diesen Daten anwendbaren Methoden.
- **funktional** (Scala, Lisp, ML, Haskell, Scheme, Erlang, Clean, F#): Programme werden als mathematische Funktionen verstanden und auch Funktionen können Daten sein. Dieses Programmierparadigma versucht, sich möglichst weit von der Architektur des Computers zu lösen. Veränderbare Speicherzellen gibt es in rein funktionalen Sprachen nicht und erst recht keine Zuweisungsbefehle.
- **Skriptsprachen** (Javascript, Perl, AWK): solche Sprachen sind dazu entworfen, einfache kleine Programme schnell zu erzeugen. Sie haben meist kein statisches Typsystem und nur eine begrenzte Zahl an Strukturierungsmöglichkeiten, oft aber eine mächtige Bibliothek, um Zeichenketten zu manipulieren.
- **logisch** (Prolog): aus der KI (künstlichen Intelligenz) stammen logische Programmiersprachen. Hier wird ein Programm als logische Formel, für die ein Beweis gesucht wird, verstanden.

Eine weitere Unterscheidung von Programmiersprachen kann in der Art des Ausführungsmodells getroffen werden. Der Programmierer schreibt den lesbaren Quelltext seines Programms. Um ein Programm auf einem Computer laufen zu lassen, muss es erst in einen Programmcode übersetzt werden, den der Computer versteht. Für diesen Schritt gibt es auch unterschiedliche Modelle:

- **kompiliert** (C, C++, Cobol, Fortran): in einem Übersetzungsschritt wird aus dem Quelltext direkt das ausführbare Programm erzeugt, das dann unabhängig von irgendwelchen Hilfen der Programmiersprache ausgeführt werden kann.
- **interpretiert** (Lisp, Scheme, Javascript): der Programmtext wird nicht in eine ausführbare Datei übersetzt, sondern durch einen Interpreter Stück für Stück anhand des Quelltextes ausgeführt. Hierzu muss stets der Interpreter zur Verfügung stehen, um das Programm auszuführen. Interpretierte Programme sind langsamer in der Ausführung als übersetzte Programme.
- **abstrakte Maschine über *byte code*** (Java, Scala, ML): dieses ist quasi eine Mischform aus den obigen zwei Ausführungsmodellen. Der Quelltext wird übersetzt in Befehle nicht für einen konkreten Computer, sondern für eine abstrakte Maschine. Für diese abstrakte Maschine steht dann ein Interpreter zur Verfügung. Der Vorteil ist, dass durch die zusätzliche Abstraktionsebene der Übersetzer unabhängig von einer konkreten Maschine Code erzeugen kann und

das Programm auf auf allen Systemen laufen kann, für die es einen Interpreter der abstrakten Maschine gibt.

Wir benützen in diesem Modul die Programmiersprache Java aus mehreren pragmatischen Gründen:

- Java ist objektorientiert und das objektorientierte Paradigma soll in diesem Modul vermittelt werden (dieses spricht zum Beispiel gegen Sprachen wie C, Pascal, Haskell).
- Java ist im Vergleich relativ aufgeräumt und die Anzahl der Konzepte noch recht übersichtlich, so dass Java didaktisch zum Unterrichten gut geeignet ist (dieses spricht zum Beispiel gegen Sprachen wie C++ oder Scala).
- Java wird in der Praxis viel eingesetzt und ist für alle gängigen Plattformen geeignet. Dabei findet sich heutzutage Java nicht nur für Desktop und Server-Anwendungen sowohl auf Windows Betriebssystemen als auf Linux-Systemen, sondern auch für die Programmierung von Android Smartphones oder Webanwendungen mit Google Web Tool (dieses spricht zum Beispiel gegen Sprachen wie Objective C oder C#.)

Man kann trefflich über die Vor- und Nachteile unterschiedlicherer Sprachen streiten, sollte aber bei der Wahl der Programmiersprache stets pragmatisch sein und sich nicht zu sehr auf emotionale Diskussionen einlassen. Streits über unterschiedliche Programmiersprachen sind müßig. Jeder hat aus unterschiedlichen Gründen Favoriten, aber ein guter Informatiker sollte sich nach etwas Einarbeitung in jeder Sprache zurecht finden. Insbesondere sollte man sich bei der Wahl der Programmiersprache von dem Einsatzgebiet der Software leiten lassen. [2]

1.3 Arbeiten mit der Kommandozeile

Wahrscheinlich ist das Hauptunterscheidungsmerkmal zwischen einem Endanwender und einem Informatiker, dass der Informatiker mit der Kommandozeile seines Betriebssystems umgehen kann. Seit Anfang der 80er Jahre gibt es graphische Benutzeroberflächen, die den Umgang mit dem Computer für Endanwender vereinfacht hat. Voreiter war hier sicher zum einem die Firma Apple mit ihren Macintosh Rechnern aber auch andere Systeme stellten in den 80er Jahren intuitive graphische Benutzeroberflächen zur Verfügung, wie der Atari ST oder Commodore Amiga. Ebenso bekamen in der Zeit Unix Betriebssysteme komfortable graphische Benutzeroberflächen wie Suns Solaris und HP-UX.

Vor dieser Zeit war eine Benutzerinteraktion immer rein über die Tastatur und fast immer über eine Kommandozeile. Bis heute stellen alle Betriebssysteme einen

[2] Und wer es wissen will: für eigene kleine Projekte benutze ich persönlich Scala oder Haskell, für Webanwendungen GWT und für mein Smartphone Java auf Android.

direkten textuellen Zugang über eine Kommandozeile zur Verfügung. Eine Kommandozeile ist ein Programm, das Texteingaben erwartet und diese für bestimmte Aktionen im Betriebssystem ausführt. Hierzu gehört zum Beispiel, dass man über die Kommandozeile beliebige Programme starten kann. Auch Programme, die selbst wieder eine graphische Benutzeroberfläche haben. Alle Programme können aber auch selbst textuelle Ausgaben auf die Kommandozeile schreiben und Texteingaben von der Kommandozeile lesen.

Die Kommandozeile ist also in der Regel nicht die direkte Interaktion mit dem Betriebssystem, sondern selbst ein Programm, das die Benutzereingaben interpretiert und in Betriebssystembefehle umsetzt. Ein solches Terminalprogramm wird als *shell* bezeichnet. Wir werden im Folgenden einen rudimentären Umgang mit dem Programm *bash* zeigen. *bash* steht für *GNU Bourne-Again SHell*. Für Linux steht die *bash* standardmäßig als Kommandozeilenprogramm zur Verfügung. Auf Windows-Systemen gibt es Software-Pakete, die installiert werden können, so dass genauso auf Windows mit *bash* gearbeitet werden kann, wie auf Linux, zum Beispiel mit www.cygwin.com (`http://www.cygwin.com/`) .

Der Vorteil des Umgangs mit der Kommandozeile ist vielfältig. Es lassen sich zum einen hier viele Aufgaben unabhängig von anderen graphischen Programmen durch Eingabe weniger Befehle bewerkstelligen. Der Zugang zum Betriebssystem ist direkter. Dinge die einem graphische Programme nicht zeigen sondern verstecken, können eingesehen werden. Es lassen sich leicht Arbeitsschritte durch Skripte automatisieren, was bei graphischen Programmen so nicht möglich ist. Und vor allem Dingen, selbst wenn keine graphische Benutzeroberfläche mehr vorliegt, lassen sich mit der Kommandozeile alle Funktionen des Systems kontrollieren. Besonders schnell erkennt man den Nutzen der Kommandozeilen, wenn man einen Server über das Internet administrieren muss. Ein Zugang zu einem Server steht über eine Kommandozeile immer zur Verfügung, eine graphische Benutzeroberfläche jedoch so gut wie nie.

Also gibt es Gründe genug, den Umgang mit Kommandozeilenbefehlen einzuüben und diesen Umgang im Laufe des Studiums immer weiter zu verbessern. Zum Glück ist es gar nicht so schwer und die wichtigsten Befehle sind schnell erlernt.

1.3.1 Basisbefehle

Wenn man die Kommandozeile geöffnet hat, befindet diese sich immer in einem Ordner des Dateisystems. Alle Befehle, die eingegeben werden beziehen sich auf diesem Ordner. Er wird auch als Arbeitsverzeichnis bezeichnet.

`ls` Dateien Auflisten

Der erste Befehl, den wir vorstellen, dient dazu die Dateien des Arbeitsverzeichnisses aufzulisten. Er besteht nur aus zwei Buchstaben: `ls`. `ls` wird als *list* gesprochen.

Geben wir den Befehl in der Kommandozeile aus, so bekommen wir eine Auflistung aller Dateien des Arbeitsverzeichnisses.

```
panitz@ThinkPad-T430:~/oose$ ls
panitz@ThinkPad-T430:~/oose$
```

In diesem Fall war die Auflistung etwas enttäuschend, denn es befindet sich keine Datei im Arbeitsverzeichnis. Wird der Befehl in einem anderen Arbeitsverzeichnis aufgerufen, in dem sich Dateien befinden, so werden diese tatsächlich aufgelistet:

```
panitz@ThinkPad-T430:~/fh/oose/v250107$ ls
build   build.properties~  build.xml~  docs  web
build.properties  build.xml      dist  src
panitz@ThinkPad-T430:~/fh/oose/v250107$
```

Es werden in diesem Fall 9 Dateien aufgelistet. Wenn wir mehr Informationen über diese Dateien erhalten wollen, so können wir dem Befehl `ls` einen zusätzlichen Befehlsparameter mitgeben. Solche beginnen in der Regel mit einem Minuszeichen. `ls` kennt den Parameter `-l`, der für *long* steht und eine ausführliche Auflistung der Dateien bewirkt.

```
panitz@ThinkPad-T430:~/fh/oose/v250107$ ls -l
insgesamt 60
drwxr-xr-x 3 panitz panitz  4096 Okt  6  2010 build
-rw-r--r-- 1 panitz panitz   253 Okt  6  2010 build.properties
-rw-r--r-- 1 panitz panitz   246 Okt  6  2010 build.properties~
-rw-r--r-- 1 panitz panitz 16334 Okt  6  2010 build.xml
-rw-r--r-- 1 panitz panitz 16336 Okt  6  2010 build.xml~
drwxr-xr-x 3 panitz panitz  4096 Okt  6  2010 dist
drwxr-xr-x 2 panitz panitz  4096 Okt  6  2010 docs
drwxr-xr-x 2 panitz panitz  4096 Okt  6  2010 src
drwxr-xr-x 3 panitz panitz  4096 Okt  6  2010 web
panitz@ThinkPad-T430:~/fh/oose/v250107$
```

Zusätzlich werden jetzt zu jeder Datei eine ganze Reihe weiterer Informationen angezeigt. Für jede Datei wird dazu eine Zeile ausgegeben. Zusätzliche Informationen sind die Benutzerrechte der Datei, die Dateigröße, das Datum der letzten Änderung und die Angabe, ob es sich bei der Datei um einen Ordner handelt.

Es gibt Dateien, die der Befehl `ls` normaler Weise nicht anzeigt, die sogenannten versteckten Dateien. Eine Datei gilt als versteckt, wenn ihr Name mit einem Punkt beginnt. Fügt man dem Befehl `ls` den Paramter `-a` hinzu, werden alle, auch die versteckten Dateien, aufgelistet.

```
panitz@ThinkPad-T430:~/oose$ ls
panitz@ThinkPad-T430:~/oose$ ls -a
.  ..  .versteckt
panitz@ThinkPad-T430:~/oose$ ls -a -l
insgesamt 20
drwxrwxr-x   2 panitz panitz  4096 Sep 18 11:27 .
drwxr-xr-x 286 panitz panitz 16384 Sep 16 14:16 ..
-rw-rw-r--   1 panitz panitz     0 Sep 18 11:27 .versteckt
panitz@ThinkPad-T430:~/oose$
```

Wie man hier sieht, wurde durch `ls` keine Datei aufgelistet. Der zusätzliche Parameter `-a` bewirkt, dass drei zusätzliche Dateien aufgelistet werden, die jeweils mit einem Punkt im Namen beginnen. `-a -l` zeigt, dass zwei dieser versteckten Dateien Ordner sind.

Auf der ursprünglich aus DOS stammenden Kommandozeile der Microsoft Betriebssysteme heißt der analoge Befehl zum Auflisten der Dateien `dir`.

`cd` Wechseln des Arbeitsverzeichnisses

Wir haben festgestellt, dass sich alle Befehle der Kommandozeile auf das Arbeitsverzeichnis beziehen. Da man natürlich nicht immer mit den Dateien eines festen Arbeitsverzeichnisses arbeiten möchte, gibt es einen Befehl zum Wechseln des Arbeitsverzeichnisses. Dieses ist der Befehl `cd`, der für *change directory* steht. Er bekommt als Argument den Ordner angegeben, in den man wechseln möchte.

```
panitz@ThinkPad-T430:~$
panitz@ThinkPad-T430:~$ cd oose
panitz@ThinkPad-T430:~/oose$
```

In diesen Beispiel wird in den Unterordner `oose` des aktuellen Arbeitsverzeichnisses gewechselt. Anschließend ist dieser das neue Arbeitsverzeichnis. Im Prompt der Kommandozeile, das ist die Meldung, mit der eine Eingabe erwartet wird, wird angezeigt, welches das aktuelle Arbeitsverzeichnis ist.

Es ist natürlich nicht nur möglich in einen bestimmten Unterordner zu wechseln, sondern man kann auch weiter in den übergeordneten Ordner wechseln. Dieser Ordner hat einen reservierten Namen, aus zwei Punkten.

```
panitz@ThinkPad-T430:~/oose$ cd ..
panitz@ThinkPad-T430:~$
```

Es gibt einen weiteren besonderen Ordner. Dieses ist das Heimatverzeichnis des Benutzers. In diesem Heimatverzeichnis startet auch standardmäßig die Kommandozeile. Dieses ist das Startverzeichnis, in dem der Benutzer seine eigenen Dateien speichert. Es wird mit dem Tildesymbol ~ bezeichnet.

```
panitz@ThinkPad-T430:~/oose$ cd ~
panitz@ThinkPad-T430:~$
```

In dieses Verzeichnis wechselt der Befehl `cd` auch, wenn kein Ordner angegeben wird, in dem gewechselt werden soll.

```
panitz@ThinkPad-T430:~/oose$ cd
panitz@ThinkPad-T430:~$
```

Es ist nicht nur möglich, relativ vom aktuellen Arbeitsverzeichnis in ein Unterverzeichnis oder das Elternverzeichnis zu wechseln, sondern man kann auch absolut von der Wurzel des Dateisystems aus einen Pfad in ein bestimmtes Verzeichnis angeben. Dann ist vollkommen egal, in welchen Arbeitsverzeichnis sich die Kommandozeile bei Ausführung des Befehls befindet. Ein absoluter Pfad beginnt mit einem Schrägstrich `/`, der die Wurzel des Gesamten Dateisystems bezeichnet. Ausgehend von dieser kann nun eine Folge von Ordner, die durch einen Schrägstrich getrennt werden.

```
panitz@ThinkPad-T430:~$ cd /home/panitz/fh/oose/v250107/
panitz@ThinkPad-T430:~/fh/oose/v250107$
```

Ein Pfad muss aber wiederum nicht absolut von der Wurzel des Dateisystems beginnen, sondern kann auch relativ vom Arbeitsverzeichnis beginnen. Hierzu ist der erste Schrägstrich wegzulassen.

```
panitz@ThinkPad-T430:~/fh/oose/v250107$ cd ../../pmt/v270513/
panitz@ThinkPad-T430:~/fh/pmt/v270513$
```

So wie es mit den zwei Punkten die Möglichkeit gibt, das übergeordnete Verzeichnis anzugeben, kann man auch das aktuelle Verzeichnis angeben. Hierzu dient der einfache Punkt.

`ls` mit Pfadangaben

Nachdem wir nun die Pfadangaben für die Kommandozeile kennen gelernt haben, können wir diese auch für den Befehl `ls` anwenden. Bisher haben wir uns von `ls` immer nur eine Auflistung der Dateien des Arbeitsverzeichnisses geben lassen. Wir können aber durch Angabe eines Pfades, der einen bestimmten Ordner bezeichnet, `ls` dazu bringen, die Dateien in diesem Ordner anzuzeigen.

```
panitz@ThinkPad-T430:~/fh/pmt/v270513$ ls /home/panitz/fh/oose/vor130605/
Counter.class GUI$3.class  Inner.java~
Counter.java GUI.class  KnopfAktion.class
Counter.java~ GUI.java  KnopfAktion.java
```

```
EverySecond.class GUI.java~  KnopfAktion.java~
EverySecond.java Inner$1.class  MyGraphic.class
EverySecond.java~ Inner$1InnerInner.class  MyGraphic.java
GUI$1.class Inner.class  MyGraphic.java~
GUI$1KnopfAktion.class Inner$InnerInner.class
GUI$2.class Inner.java
panitz@ThinkPad-T430:~/fh/pmt/v270513$
```

Sollte man nur an Informationen einer bestimmten Datei interessiert sein, so lässt sich dieses durch einen Pfad auf diese Datei,bestimmen.

```
panitz@ThinkPad-T430:~$ ls -l /home/panitz/fh/oose/vor130605/GUI.java
-rw-r--r-- 1 panitz panitz 1111 Okt  6  2010 /home/panitz/fh/oose/vor130605/GUI.java
panitz@ThinkPad-T430:~$
```

Innerhalb von Dateinamen hat der Stern `*` noch eine besondere Bedeutung. Er steht für eine beliebige Folge von beliebigen Buchstaben. So bedeutet z.B. `*.jpg` alle Dateinamen mit der Endung `.jpg`. So lassen sich Mengen von Dateien ansprechen. Im folgenden Beispiel werden alle Dateien, deren Name mit `GUI` beginnt und die die Endung `.class` haben.

```
panitz@ThinkPad-T430:~$ ls -l /home/panitz/GUI*.class
-rw-r--r-- 1 panitz panitz  875 Okt  6  2010 /home/panitz/GUI$1.class
-rw-r--r-- 1 panitz panitz  624 Okt  6  2010 /home/panitz/GUI$1KnopfAktion.class
-rw-r--r-- 1 panitz panitz  875 Okt  6  2010 /home/panitz/GUI$2.class
-rw-r--r-- 1 panitz panitz  585 Okt  6  2010 /home/panitz/GUI$3.class
-rw-r--r-- 1 panitz panitz 1655 Okt  6  2010 /home/panitz/GUI.class
panitz@ThinkPad-T430:~$
```

`mkdir` Neue Verzeichnisse Anlegen

Es lassen sich mit dem Befehl `mkdir` neue Verzeichnisse anlegen. Hierzu schriebt man hinter den Befehl als Pfadangabe den Namen des anzulegenden Verzeichnisses. Dieses kann wieder relativ zu dem Arbeitsverzeichnis sein oder auch eine absolute Pfadangabe.

```
panitz@ThinkPad-T430:~$ mkdir neuesTestVerzeichnis
panitz@ThinkPad-T430:~$ cd neuesTestVerzeichnis/
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$
```

Mit dem Befehl `rmdir` lassen sich Verzeichnisse wieder löschen. Hierzu muss das Verzeichnis allerdings leer sein. Es darf keine Unterverzeichnisse oder Dateien mehr enthalten.

```
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$ cd ..
panitz@ThinkPad-T430:~$ rmdir neuesTestVerzeichnis/
panitz@ThinkPad-T430:~$
```

`touch` Neue Dateien Anlegen oder Dateien Aktualisieren

Es lassen sich auch neue leere Dateien von der Kommandozeile anlegen. Hierzu kann der Befehl `touch` benutzt werden. Ihm gibt man an, welche Datei neu angelegt werden soll.

```
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$ ls -l
insgesamt 0
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$ touch neueTestdatei
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$ ls -l
insgesamt 0
-rw-rw-r-- 1 panitz panitz 0 Sep 19 10:33 neueTestdatei
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$
```

Sollte der Befehl für eine Datei aufgerufen werden, die bereits existiert, so wird der Dateiinhalt nicht verändert, aber das Datum der letzten Änderung aktualisiert.

```
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$ ls -l
insgesamt 0
-rw-rw-r-- 1 panitz panitz 0 Sep 19 10:33 neueTestdatei
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$ touch neueTestdatei
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$ ls -l
insgesamt 0
-rw-rw-r-- 1 panitz panitz 0 Sep 19 10:34 neueTestdatei
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$
```

`cat, more` und `less` Dateiinhalte Anzeigen

Auch der Inhalt einer Textdatei lässt sich auf der Kommandozeile anzeigen. Hierzu kann der Befehl `cat` benützt werden. `cat` steht dabei für *concatenate*, also zum Konkatenieren, dem Aneinanderhängen von Dateiinhalten. Der Name kommt daher, dass dem Befehl `cat` nicht nur eine sondern eine Liste von Dateien angegeben werden kann. Es werden dann die Inhalte dieser Dateien nacheinander auf der Kommandozeile ausgegeben.

```
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$ cat neueTestdatei
Hier steht nur ein kleiner Beispieltext.
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$
```

Ist eine Textdatei sehr groß, so flutscht die Ausgabe von `cat` sehr schnell auf der Kommandozeile vor unserem Auge. Man will einen Dateiinhalt in der Regel durchlesen. Hierzu gibt es die funktionsgleichen Programme `more` und `less`. Auch mit diesen Befehlen wird der Dateiinhalt auf der Kommandozeile ausgegeben. Allerdings immer nur ein Seite, die genau auf das Fenster der Kommandozeile passt. Nun kann man mit Drücken der Leertaste jeweils zur nächsten Seite gesprungen werden. Mit Drücken der Taste Q wird die Anzeige der Datei beendet.

`pwd` Pfadangabe

Man kann durch den Befehl `pwd` sich den absoluten Pfad zum Arbeitsverzeichnis anzeigen lassen.

```
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$ pwd
/home/panitz/neuesTestVerzeichnis
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$
```

`rm` Dateien Löschen

Dateien lassen sich auch löschen. Hierzu dient der Befehl `rm`, der memotechnisch für *remove* steht. (Auf der Kommandozeile für Microsoft Betriebssysteme heißt der entsprechende Befehl `del`.) Dieser Befehl ist nichts für Feiglinge, denn die mit ihm gelöschten Dateien sind unwiederbringlich weg. Sie sind nicht in einem Papierkorb zwischengelagert, aus dem sie wieder reaktiviert werden können. Besonders gefährlich kann es sein, wenn dieser Befehl mit einer Dateiangabe aufgerufen wird, die über das Sternsymbol mehrere Dateien bezeichnet. `rm *.jpg *.gif *.png` löscht zum Beispiel alle Bilddateien im Arbeitsverzeichnis. `rm *` löscht alle Dateien im Arbeitsverzeichnis. Der Befehl kennt dann noch den Parameter `-r`. Er steht dafür, dass im Falle eines Ordners zunächst alle Dateien und Unterordner dieses Ordners zu löschen sind und dann der Ordner selbst. So löscht also `rm -r *` alle Dateien und alle Ordner komplett aus dem Arbeitsverzeichnis. In Ihrem Heimatverzeichnis aufgerufen löschen Sie damit also alle Ihre Dateien. Also Vorsicht!

`mv` Dateien Umbenennen und Verschieben

Der Befehl `mv` steht memotechnisch für *move*. Er dient dazu, eine Datei an einen anderen Ort, also in einen anderen Ordner, zu verschieben. Er hat aber auch eine zweite verwandte Funktionalität. Er dient auch dazu eine Datei umzubenennen. Mindestens zwei Parameter sind diesem Befehl anzugeben. Die Datei, die verschoben oder umbenannt werden soll, und das Ziel wohin diese verschoben werden soll. Ist das Ziel ein existierender Ordner im Dateisystem, so wird die Datei dahin verschoben. Ist es ein Dateiname eventuell einer noch nicht existierenden Datei, dann wird die Datei umbenannt.

```
panitz@ThinkPad-T430:~/oose$ ls
testdatei.txt
panitz@ThinkPad-T430:~/oose$ mv testdatei.txt neuerName.txt
panitz@ThinkPad-T430:~/oose$ ls
neuerName.txt
panitz@ThinkPad-T430:~/oose$ mkdir einNeuerOrdner
panitz@ThinkPad-T430:~/oose$ mv neuerName.txt einNeuerOrdner/
panitz@ThinkPad-T430:~/oose$ ls
einNeuerOrdner
```

```
panitz@ThinkPad-T430:~/oose$ cd einNeuerOrdner/
panitz@ThinkPad-T430:~/oose/einNeuerOrdner$ ls
neuerName.txt
panitz@ThinkPad-T430:~/oose/einNeuerOrdner$
```

Auch hier muss man ein wenig vorsicht walten lassen. Existiert die Zieldatei bereits, so wird sie mit der Quelldatei überschrieben und damit quasi gelöscht.

Man kann den Befehl `mv` auch nutzen, um mehrere Dateien in einen Ordner zu verschieben. Dann erhält der Befehl mehr als zwei Argumente. Das letzte Argument bezeichnet dabei einen Ordner, in den die vorherigen Argumente zu verschieben sind. Auch hier kann man sich wieder des Sterns bedienen. So bewirkt `mv *.jpg meineBilder`, dass alle Dateien des Arbeitsverzeichnisses mit der Endeung `jpg` in den Unterordner `meineBilder` verschoben werden.

`cp` Dateien Kopieren

Sehr ähnlich, wie der Befehl `mv` funktioniert der Befehl `cp`, der dazu dient eine oder mehrere Dateien zu kopieren. Es handelt sich dazu im Prinzip um den gleichen Befehl wie `mv`, nur wird die Quelldatei in diesem Fall nicht gelöscht. Damit gibt es anschließend also zwei unabhängige Versionen der Quelldatei.

`man` Handbuchseiten

Wie kann man sich alle die Befehle und ihre Benutzung merken. Alle die kleinen Parameter. Die Antwort ist: RTFM. Dieses steht für den Satz *read the fucking manual* also eine etwas saloppe Aufforderung, das Handbuch zu lesen. Mit dem Befehl `man` können die Handbucheinträge aller Befehle aufgerufen werden. Wer z.B. nicht mehr weiß, wie der Befehl `ls` genau funktioniert, kann das mit `man ls` erfragen und bekommt folgende Ausgabe:

```
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$ man ls

LS(1)                            User Commands                            LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List  information  about  the FILEs (the current directory by default).
       Sort entries alphabetically if none of -cftuvSUX nor --sort  is  speci-
       fied.

       Mandatory  arguments  to  long  options are mandatory for short options
```

```
       too.

       -a, --all
              do not ignore entries starting with .

       -A, --almost-all
              do not list implied . and ..

       --author
 Manual page ls(1) line 1 (press h for help or q to quit)
```

Wie sie sehen, gibt es mehr Optionen für diesen Befehl, als in diesem Skript angegeben. Das gilt für alle der hier vorgestellten Befehle.

Es lässt sich auch die Hilfe für die gesamte Kommandozeilenbenutzung anzeigen durch `man bash`.

1.3.2 Nützliche Standardprogramme

Im letzten Abschnitt haben wir die wichtigsten Befehle der Kommandozeile zum Umgang mit dem Dateisystem kennen gelernt. Prinzipiell kann jedes Programm über die Kommandozeile gestartet werden. Viele Programme haben selbst gar keine graphische Benutzerschnittstelle sondern sind zur Benutzung über die Kommandozeile vorgesehen, oder werden von anderen Programmen aus gestartet. In diesem Abschnitt sollen ein paar elementare Programme, die sehr nützlich sind kurz vorgestellt werden.

Sicheres Einloggen auf anderen Rechner mit `ssh`

Das Programm `ssh` dient dazu, um sich auf einem Rechner über ein Netzwerk einzuloggen und quasi die dortige Kommandozeile zu benützen. Dieses ist natürlich besonders nützlich, wenn man einen Webserver warten möchte. Hierzu ist Beim Aufruf von `ssh` der Rechnername des Rechners, auf den man sich einloggen möchte, als Argument anzugeben. Wir haben bei uns im Studienbereich den Rechner mit der Adresse `login1.cs.hs-rm.de`, auf dem sie sich mit ihrem Passwort einloggen können. Um anzugeben, als welcher Benutzer Sie sich auf dem Rechner einloggen wollen, können Sie Ihren Benutzernamen mit dem Klammeraffensymbol getrennt dem Rechnernamen voranstellen.

```
panitz@ThinkPad-T430:~/oose$ ssh panitz@login1.cs.hs-rm.de
panitz@login1.cs.hs-rm.de's password:

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.
```

```
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Tue Oct  1 12:10:22 2013 from vpn501.hotsplots.net
panitz@fozzie(~)$
```

Jetzt arbeiten Sie auf dem Server. Dieses erkennen Sie zum Beispiel, wenn Sie jetzt den Befehl `ls` eintippen, denn dann stellen Sie fest, dass Sie die Dateien an der Hochschule und nicht auf Ihrem Rechner daheim sehen.

Wollen Sie die Verbindung mit dem fremden Rechner beenden, so können Sie das durch den Befehl `exit`:

```
panitz@fozzie(~)$ exit
Abgemeldet
Connection to login1.cs.hs-rm.de closed.
panitz@ThinkPad-T430:~/oose$
```

Transfer von Dateien mit `sftp`

Mit `ssh` können Sie auf die Kommandozeile eines Rechners über ein Netzwerk zugreifen. Manchmal will man aber eine oder mehrere Dateien von einem anderen Rechner holen oder auf einen anderen Rechner hoch laden. Hierzu dient das Programm `sftp` *(secure file transfer program)*. Auch dieses wird mit ihrem Benutzernamen gefolgt von der Adresse des Servers gestartet. Wenn Sie in den Server eingeloggt sind, sind sie in einer Umgebung, in der Sie Befehle zum Dateitransfer abschicken können. Die Befehle heißen `put`, um Dateien auf den Server hoch zu laden und `get` um Dateien von dem Server herunter zu laden. Mit dem aus der Kommandozeile bekannten Befehl `cd` können Sie durch das Dateisystem des Servers navigieren.

```
panitz@ThinkPad-T430:~/oose/einNeuerOrdner$ sftp panitz@login1.cs.hs-rm.de
panitz@login1.cs.hs-rm.de's password:
Connected to login1.cs.hs-rm.de.
sftp> put existierendeDatei
Uploading existierendeDatei to /home/staff/panitz/existierendeDatei
existierendeDatei                             100%    0     0.0KB/s   00:00
sftp> get Main.hs
Fetching /home/staff/panitz/Main.hs to Main.hs
sftp> exit
panitz@ThinkPad-T430:~/oose/einNeuerOrdner$
```

Auch dieses Programm wird durch die Eingabe des Befehls `exit` wieder beendet.

Dateityp erfragen mit `file`

Eigentlich sollte der Typ einer Datei durch die Dateiendung korrekt angegeben sein. Manchmal fehlt aus unerfindlichen Gründen diese Endung, oder aber, Sie wurde falsch angegeben (Das passiert immer wieder bei Abgaben von studentischen Lösungen). Oder aber man kennt die Endung nicht. Dann ist das Programm `file` recht nützlich, denn es versucht herauszubekommen, um was für eine Datei es sich handelt.

```
panitz@ThinkPad-T430:~$ file FKT_Programm_11_13_V1_Montageflaechen-1
FKT_Programm_11_13_V1_Montageflaechen-1: PDF document, version 1.4
panitz@ThinkPad-T430:~$
```

Dateien in ein Archiv verpacken mit `tar`

Zur Archivierung oder dem Versand von einer großen Menge von Dateien, zum Beispiel aller Ihrer Urlaubsbilder, empfiehlt es sich, diese in einer einzigen Archivdatei zu bündeln und am besten noch zu komprimieren. Hierzu gibt es mehrere Programme und daraus resultierende Dateiformate wie zum Beispiel das Programm `zip`. Ein Standardprogramm auf der Unix-Welt ist hierzu das Programm `tar`. Es steht für *type archive* und deutet so im Namen auf die Zeiten hin, als Daten auf Tonbändern archiviert wurden. Dem Programm `tar` wird durch eine Folge vom Buchstaben beim Aufruf angegeben, was es machen soll, z.B. ob es ein neues Archiv erzeugen oder ob es ein bestehendes entpacken soll. Dann folgen die Dateien, mit denen gearbeitet werden soll. das Erzeugen eines neuen Archivs wird durch den Buchstaben `c` für *create* angegeben. Mit dem Buchstaben `f` wird angegeben, dass jetzt der Name der Archivdatei folgt. Will man also alle Bilddateien im Arbeitsverzeichnis in eine Archivdatei `bilder.tar` verpacken, so geht das mit dem Befehl `tar cf bilder.tar *.jpg`.

Zum Entpacken einer Archivdatei benützt man den Buchstaben `x` für *extract* statt des Buchstabens `c`. Die mit obigen Befehle erzeugte Archivdatei lässt sich also mit `tar xf bilder.tar` wieder entpacken.

Will man sich nur eine Auflistung der Dateien, die in einer Archivdatei verpackt wurden, geben lassen, so nimmt man den Buchstaben `t` für *table*. Fügt man noch den Buchstaben `v` für *verbode* hinzu, so gibt das Programm etwas mehr Ausgaben. Soll die Archivdatei komprimiert werden, so füge man noch den Buchstaben `z` hinzu.

Hier ein kleines Beispiel zum Verpacken, Auflisten und Entpacken einer kleinen Archivdatei:

```
panitz@ThinkPad-T430:~/Bilder$ tar cvzf bilder.tgz *.png
Bildschirmfoto vom 2013-07-17 19:05:30.png
Bildschirmfoto vom 2013-08-01 19:49:16.png
Bildschirmfoto vom 2013-08-01 21:59:44.png
Logo-gross.png
```

```
tree.png
panitz@ThinkPad-T430:~/Bilder$ tar tvzf bilder.tgz
-rw-rw-r-- panitz/panitz 368235 2013-07-17 19:05 Bildschirmfoto vom 2013-07-17 19:05:30.png
-rw-rw-r-- panitz/panitz 157235 2013-08-01 19:49 Bildschirmfoto vom 2013-08-01 19:49:16.png
-rw-rw-r-- panitz/panitz 156051 2013-08-01 21:59 Bildschirmfoto vom 2013-08-01 21:59:44.png
-rw-r--r-- panitz/panitz  14209 2009-01-03 14:47 Logo-gross.png
-rw-rw-r-- panitz/panitz   3733 2013-05-14 15:25 tree.png
panitz@ThinkPad-T430:~/Bilder$ tar xvzf bilder.tgz
Bildschirmfoto vom 2013-07-17 19:05:30.png
Bildschirmfoto vom 2013-08-01 19:49:16.png
Bildschirmfoto vom 2013-08-01 21:59:44.png
Logo-gross.png
tree.png
panitz@ThinkPad-T430:~/Bilder$
```

Weitere Tipps und Tricks

Den ersten und rudimentären Umgang mit der Kommandozeile haben wir eingeübt.

Arbeiten mit Vervollständigung und Befehlshistorie Die Kommandozeile merkt sich die bereits ausgeführten Befehle. Oft will man einen ähnlichen oder den gleichen Befehl noch einmal ausführen lassen. Die Pfeiltasten liefern heute eine einfache Möglichkeit durch die Befehlshistorie zu navigieren und zuvor bereits ausgeführte Befehle wieder anzeigen zu lassen.

Auch die Tabulaturtaste vereinfacht die Arbeit mit der Kommandozeile und hilft dabei, weniger eintippen zu müssen. Wenn von der Kommandozeile der Name einer Datei erwartet wird, kann man die ersten Buchstaben der Datei tippen und dann mit Hilfe der Tabulatortaste sich den kompletten Dateinamen vervollständigen zu lassen. So müssen lange Dateinamen nicht komplett eingetippt werden.

Umleiten der Ausgabe in eine Datei Wenn man einen Befehl startet, der eine ausführliche Ausgabe auf der Kommandozeile macht, dann kann man diese auf einfache Weise von der Kommandozeile in eine Datei schreiben lassen. Hierzu bedient man sich des Größer-Zeichens. Nach dem Befehl wird das Größer-Zeichen und der Dateiname geschrieben. Dieses bewirkt, dass die Ausgabe des Befehls nicht mehr auf der Kommandozeile angezeigt wird, sondern in die angegebene Datei geschrieben wird.

Zwei Programme verbinden Ein besonders eleganter Trick ist es, die Ausgabe eines Befehls direkt wieder als Eingabe für einen weiteren Befehl zu benützen. Hierzu dient die sogenannte *pipe*, die durch einen vertikalen Strich bezeichnet wird.

Ist das Arbeitsverzeichnis zum Beispiel ein Ordner, in dem sehr viele Dateien liegen, dann führt der Befehl `ls -l` zu einer umfangreichen Ausgabe auf der Kommandozeile, die schnell auf dem Bildschirm vorbei rauscht. Jetzt kann man die Ausgabe

von `ls -l` direkt an das Programm `less` weiterleiten, indem man die Ausgabe von `ls` dem Befehl `less` über eine *pipe* direkt als Eingabe gibt. Der gesamte Aufruf lautet dann `ls -l | less`.

1.3.3 Erste Java Programme auf der Kommandozeile

In diesem Abschnitt soll gezeigt werden, wie prinzipiell Programme auf der Kommandozeile entwickelt werden. Wir erinnern uns, dass ein Programm zunächst einmal nichts weiter als ein Text ist. Wir müssen also einen Text schreiben. Dieser Text ist dann von einem besonderen Programm in eine Maschinensprache zu Übersetzen. Ein solche Programm heißt Kompilator *(eng. compiler)*. Der Kompilator untersucht den Quelltext und übersetzt diesen, sofern das möglich ist, in eine oder mehrere binäre Dateien, die Befehle für eine Maschine codieren. Schließlich müssen diese binären Dateien zur Ausführung auf einer Maschine gebracht werden.

Quelltext editieren

Wir benötigen ein Programm, dass uns erlaubt eine Quelltextdatei zu erstellen. Ein solches Programm nennt man einen Texteditor. Dieser ist nicht zu verwechseln mit einem Textverarbeitungsprogramm, in dem nicht nur Text, sondern auch ein Layout mit Schriftarten, Schriftgrößen etc erstellt wird. Es gibt tatsächlich Programme, die dieses alleine auf der Kommandozeile erlauben. Der Standardeditor für die Kommandozeile heißt `vi`. Es gibt sogar auch heute noch Leute, die nur mit Hilfe des `vi` ihre Programme schreiben. Aber gerüchteweise sollen das nur noch fusselige, bärtige, nuschelnde, selbstdrehende Systemadministratoren sein.

trotzdem sollt jeder Informatiker den rudimentären Umgang mit dem Programm `vi` kennen, denn irgendwann in seiner Laufbahn wird der Moment kommen, an dem er schnell eine Konfigurationsdatei ändern muss, aber nur einen Kommandozeilenzugang zu dem Rechner hat.

Das Programm `vi` kann gestartet werden mit dem Namen der Datei, die man editieren möchte. Existiert keine Datei mit diesen Namen, so wird die Datei dann neu angelegt. Wenn wir also eine Javaquelltextdatei mit dem Dateinamen `FirstProgra,.java` editieren und erstellen möchten, so starten wir den `vi` mit dem Befehl:

```
vi FirstProgram.java
```

Das Programm `vi` hat zwei Modi. Den Befehlsmodus, in dem jede Eingabe als Befehl für den Editor interpretiert wird, und den Einfügemodus, in dem jede Eingabe als Text für den Inhalt der Datei interpretiert wird. Der Befehlsmodus ersetzt die Menus, die aus den Texteditoren mit graphischer Benutzerführung bekannt sind.

Vom Befehlsmodus kann man in den Einfügemodus durch Eingabe von `i`für *insert* oder `a` für *append* wechseln. Nach der Eingabe des Befehls `i` befindet man sich in Einfügemodus und alle folgenden Buchstaben werden vor der markierten Stelle des Dokuments eingefügt. Nach dem Befehl `a` werden beim Einfügemodus alle folgenden Buchstaben nach der ursprünglich markierten Stelle eingefügt.

Vom Einfügemodus in den Befehlsmodus wechselt man durch Drücken der `ESC`-Taste.

Die wichtigsten Befehlsfolgen im Befehlsmodus betreffen natürlich das Speichern der Datei und auch das Verlassen des Programms. Diese Befehlsfolgen werden durch den Doppelpunkt eingeleitet. Der Befehl `:wq` bewirkt im Befehlsmodus, dass die Datei gespeichert wird (`w` für *write*) und das Programm `vi` verlassen wird (`q` für *quit*).

Versuchen Sie also Ihre erste Javaquelltextdatei mit dem `vi` zu schreiben. Jede Javaquelltextdatei hat die Endung `.java`. In einer Javaquelltextdaei wird jeweils genau eine sogenannte Klasse definiert. Folgendes ist die kleinste mögliche Klasse. Sie heißt `FirstProgram` und ist entsprechend in eine Datei `FirstProgramm.java` zu speichern.

```
1 class FirstProgramm{
2 }
```

Listing 1.1: FirstProgramm.java

Es gibt noch eine ganze Reihe weiterer Texteditoren, die auf der Kommandozeile benutzt werden können. Ich persönlich benutze, wann immer möglich das Programm `emacs`, welche mit einer Menuführung daherkommt, aber auch einen Modus hat, der ohne graphische Benutzeroberfläche auskommt. Hierzu startet man den `emacs` mit dem Argument `-nw` (für *no window*).

Ein weiteres solcher Texteditor ist das Programm `joe`. Allerdings sind `emacs` oder `joe` nicht auf allen Systemen installiert, wohingegen man eigentlich immer davon ausgehen kann, dass `vi` existiert.

Quelltext kompilieren

Nun brauchen wir den Kompilator, der einen Quelltext in einen binären Maschinencode überführt. Für Java heißt das entsprechende Programm `javac`. Durch den Aufruf des Programms `javac` kann man sich davon überzeugen, dass die Java-Entwicklerumgebung auf dem Rechner installiert ist. Ruft man den Kompilator auf, so macht er folgende Ausgabe, die über seine Benutzung informiert:

```
panitz@ThinkPad-T430:~$ javac
Usage: javac <options> <source files>
where possible options include:
  -g                         Generate all debugging info
```

```
  -g:none                      Generate no debugging info
  -g:{lines,vars,source}       Generate only some debugging info
  -nowarn                      Generate no warnings
  -verbose                     Output messages about what the compiler is doing
  -deprecation                 Output source locations where deprecated APIs are used
  -classpath <path>          Specify where to find user class files and annotation processors
  -cp <path>                 Specify where to find user class files and annotation processors
  -sourcepath <path>           Specify where to find input source files
  -bootclasspath <path>        Override location of bootstrap class files
  -extdirs <dirs>              Override location of installed extensions
  -endorseddirs <dirs>         Override location of endorsed standards path
  -proc:{none,only}         Control whether annotation processing and/or compilation is done.
  -processor <class1>[,<class2>,<class3>...] Names of the annotation processors to run; bypasses defaul
  -processorpath <path>        Specify where to find annotation processors
  -d <directory>               Specify where to place generated class files
  -s <directory>               Specify where to place generated source files
  -implicit:{none,class}     Specify whether or not to generate class files for implicitly referenced fil
  -encoding <encoding>         Specify character encoding used by source files
  -source <release>            Provide source compatibility with specified release
  -target <release>            Generate class files for specific VM version
  -version                     Version information
  -help                        Print a synopsis of standard options
  -Akey[=value]                Options to pass to annotation processors
  -X                           Print a synopsis of nonstandard options
  -J<flag>                     Pass <flag> directly to the runtime system
  -Werror                      Terminate compilation if warnings occur
  @<filename>                  Read options and filenames from file

panitz@ThinkPad-T430:~$
```

Wie man dieser Ausgabe entnehmen kann, gibt es das Argument `-version`, mit dem die Versionsnummer des Kompilators erfragt werden kann.

```
panitz@ThinkPad-T430:~$ javac -version
javac 1.7.0_21
panitz@ThinkPad-T430:~$
```

In diesem Fall handelt es sich also um den Java-Kompilator für Java der Version 1.7.

Jetzt können wir für die Java-Quelltextdatei `FirstProgram.java` durch den Kompilator in eine binäre Datei übersetzen lassen. Hierzu wird der Kompilator mit dem Dateinamen der Quelltextdatei als Argument aufgerufen. Das Ergebnis ist im Erfolgsfall eine neu generierte Datei, die `class`-Datei.

```
panitz@ThinkPad-T430:~/oose$ ls
FirstProgram.java
panitz@ThinkPad-T430:~/oose$ javac FirstProgram.java
panitz@ThinkPad-T430:~/oose$ ls
FirstProgram.java  FirstProgram.class
panitz@ThinkPad-T430:~/oose$
```

Wie man sieht wurde die Quelltextdatei erfolgreich übersetzt und eine Class-Datei erzeugt. Sehr oft ruft man den Kompilator auf, doch statt den Quelltext zu übersetzen, bricht er mit einer Fehlermeldung ab. Dieses ist der Fall, wenn in der Quelltextdatei kein gültiges Java-Programm geschrieben ist. In unseren Fall war aber der Übersetzungsvorgang erfolgreich.

Programme ausführen

Schließlich soll die binäre Datei mit den Maschinenbefehlen, ausgeführt werden. Im Falle von Java handelt es sich dabei nicht um eine real als Hardware existierende Maschine, sondern um eine gedachte, eine sogenannte virtuelle Maschine. Dieses wird durch ein Programm realisiert. Daher braucht man zum Ausführen von Javaprogrammen das Programm, das die virtuelle Maschine realisiert. Dieses Programm heißt sinniger Weise `java`. Auch in diesem Fall kann man sich durch den Befehl `java` auf der Kommandozeile davon überzeugen, dass die virtuelle Maschine auf dem Rechner installiert ist. Das Programm `java` wird auch als Javainterpreter bezeichnet.

Der Javainterpreter wird aufgerufen mit dem Namen der Klasse, die ausgeführt werden soll. Hierbei wird kein Dateiname angegeben, auch keine Dateiendung, sondern nur der Name der Klasse, in unserem Fall `FirstProgram`. Somit ist der Befehl zum Ausführen der Klasse `java FirstProgram`.

Rufen wir dieses auf, so stellen wir fest, dass der Javainterpreter eine Fehlermeldung ausgibt:

```
panitz@ThinkPad-T430:~/oose$ java FirstProgram
Fehler: Hauptmethode in Klasse FirstProgram nicht gefunden. Definieren Sie die Hauptmethode als:
   public static void main(String[] args)
panitz@ThinkPad-T430:~/oose$
```

In diesem Falll ist die Meldung sogar auf Deutsch. Sie besagt, dass der Klasse etwas fehlt, nämlich eine sogenannte Hauptmethode, in der die Javabefehle stehen, die ausgeführt werden sollen. Die Fehlermeldung gibt sogar an, wie eine solche Hauptmethode auszusehen hat. Schreiben wir jetzt einmal eine zweite Klasse, die eine solche Hauptmethode beinhaltet:

```
1 class SecondProgram{
2   public static void main(String[] args){
3   }
4 }
```

Listing 1.2: SecondProgram.java

Übersetzen wir diese Klasse mit dem Kompilator und interpretieren sie mit dem Javainterpreter, so gibt es keine Fehlermeldung mehr. Allerdings passiert auch

nichts, weil der innerhalb der Hauptmethode keine Befehle stehen, die ausgeführt werden sollen.

```
panitz@ThinkPad-T430:~/oose$ javac SecondProgram.java
panitz@ThinkPad-T430:~/oose$ java SecondProgram
panitz@ThinkPad-T430:~/oose$
```

Der erste und einfachste Befehl, mit dem man ein Programm dazu bringen kann, eine Rückmeldung an den Anwender zu geben, ist die Ausgabe eines Textes auf der Kommandozeile. In Java ist dieser Befehl: `System.out.println()`. In die runden Klammern ist der text zu schreiben, der ausgegeben werden soll. Text wird dabei in Java in doppelten Anführungszeichen gesetzt.

Die folgende Javaklasse hat eine Hauptmethode innerhalb derer zweimal eine Zeile mit Text auf die Kommandozeile ausgegeben wird:

```
1 class ThirdProgram{
2   public static void main(String[] args){
3     System.out.println("Hallo Ilja!");
4     System.out.println("Hallo Welt!");
5   }
6 }
```

Listing 1.3: ThirdProgram.java

Kompilieren wir diese Klasse und interpretieren sie, so haben wir folgendes Verhalten:

```
panitz@ThinkPad-T430:~/oose$ javac ThirdProgram.java
panitz@ThinkPad-T430:~/oose$ java ThirdProgram
Hallo Ilja!
Hallo Welt!
panitz@ThinkPad-T430:~/oose$
```

Aufgabe 1 Entwickeln Sie Ihre erste minimale Javaklasse nur mit Hilfe der Kommandozeile. Benutzen Sie einen einfachen Editor zum Editieren, den Javakompilator zum Übersetzen und die virtuelle Maschine von Java zum Ausführen.

Kapitel 2

Grundkonzepte der Objektorientierung

Die Grundidee der objektorientierten Programmierung ist, Daten, die zusammen ein größeres zusammenhängendes Objekt beschreiben, zusammenzufassen. Zusätzlich fassen wir mit diesen Daten noch die Programmteile zusammen, die diese Daten manipulieren. Ein Objekt enthält also nicht nur die reinen Daten, die es repräsentiert, sondern auch Programmteile, die Operationen auf diesen Daten durchführen. Insofern wäre vielleicht *subjektorientierte Programmierung* ein passenderer Ausdruck, denn die Objekte sind nicht passive Daten, die von außen manipuliert werden, sondern enthalten selbst als integralen Bestandteil Methoden, die ihre Daten manipulieren können.

2.1 Objektorientierte Modellierung

Bevor wir etwas in Code gießen, wollen wir erst einmal eine informelle Modellierung der Welt, für die ein Programm geschrieben werden soll, vornehmen. Hierzu empfiehlt es sich durchaus, in einem Team zusammenzusitzen und auf Karteikarten aufzuschreiben, was es denn für Objekte in der Welt gibt, die wir modellieren wollen.

Stellen wir uns hierzu einmal vor, wir sollen ein Programm zur Bibliotheksverwaltung schreiben. Jetzt überlegen wir einmal, was gibt es denn für Objektarten, die alle zu den Vorgängen in einer Bibliothek gehören. Hierzu fällt uns vielleicht folgende Liste ein:

- Personen, die Bücher ausleihen wollen.
- Bücher, die ausgeliehen werden können.
- Tatsächliche Ausleihvorgänge, die ausdrücken, dass ein Buch bis zu einem bestimmten Zeitpunkt von jemanden ausgeliehen wurde.
- Termine, also Objekte, die ein bestimmtes Datum kennzeichnen.

Nachdem wir uns auf diese vier für unsere Anwendung wichtigen Objektarten geeinigt haben, nehmen wir vier Karteikarten und schreiben jeweils eine der Objektarten als Überschrift auf diese Karteikarten.

Jetzt haben wir also Objektarten identifiziert. Im nächsten Schritt ist zu überlegen, was für Eigenschaften diese Objekte haben. Beginnen wir für die Karteikarte, auf der wir als Überschrift *Person* geschrieben haben. Was interessiert uns an Eigenschaften einer Person? Wahrscheinlich ihr Name mit Vorname, Straße und Ort sowie Postleitzahl. Das sollten die Eigenschaften einer Person sein, die für ein Bibliotheksprogramm notwendig sind. Andere mögliche Eigenschaften wie Geschlecht, Alter, Beruf oder ähnliches interessieren uns in diesem Kontext nicht. Jetzt schreiben wir die Eigenschaften, die uns von einer Person interessieren, auf die Karteikarte mit der Überschrift Person.

Schließlich müssen wir uns Gedanken darüber machen, was diese Eigenschaften eigentlich für Daten sind. Name, Vorname, Straße und Wohnort sind sicherlich als Texte abzuspeichern oder, wie der Informatiker gerne sagt, als Zeichenketten. Die Postleitzahl ist hingegen als eine Zahl abzuspeichern. Diese Art, von der die einzelnen Eigenschaften sind, nennen wir ihren Typ. Wir schreiben auf die Karteikarte für die Objektart *Person* hinter jede der Eigenschaften noch den Typ, den diese Eigenschaft hat. Damit erhalten wir für die Objektart *Person* die in Abbildung 2.1 gezeigte Karteikarte.

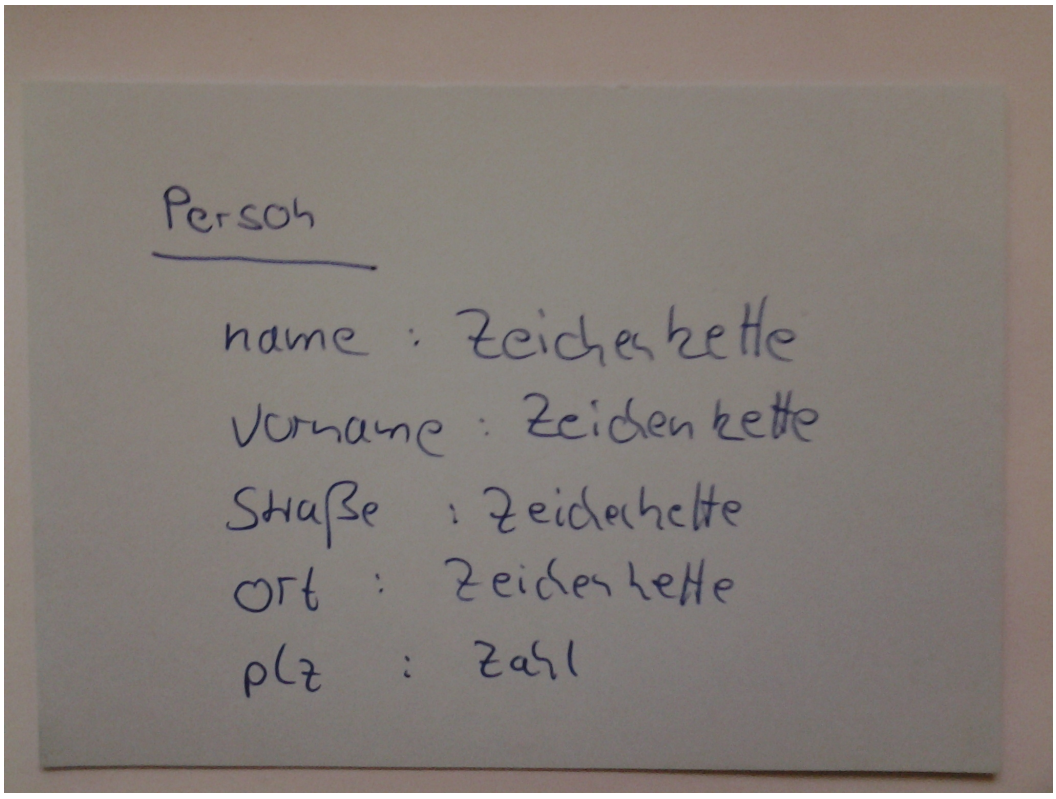

Abbildung 2.1: Modellierung einer Person.

Gleiches können wir für die Objektart *Buch* und für die Objektart *Datum* machen. Wir erhalten dann die Karteikarten aus Abbildung **??** und 2.3 .

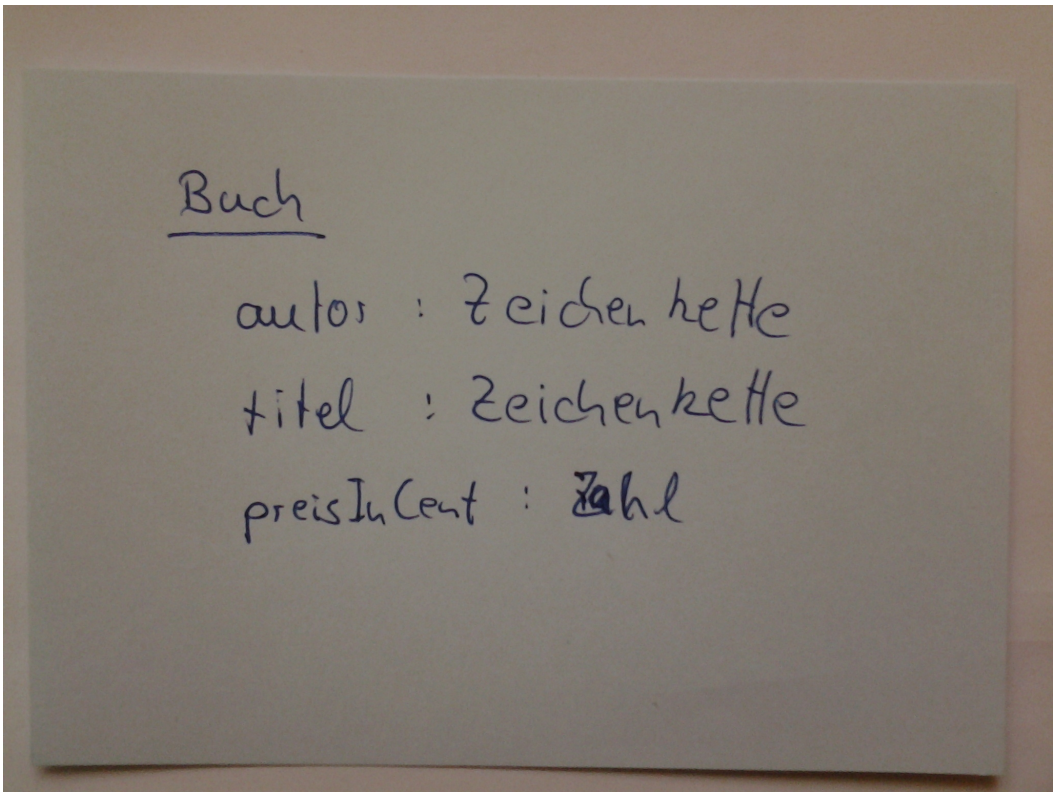

Abbildung 2.2: Modellierung eines Buches.

Wir müssen uns schließlich nur noch um die Objektart einer Buchausleihe kümmern. Hier sind drei Eigenschaften interessant: wer hat das Buch geliehen, welches Buch wurde verliehen und wann muss es zurückgegeben werden. Wir können also drei Eigenschaften auf die Karteikarte schreiben. Was sind die Typen dieser drei Eigenschaften? Diesmal sind es keine Zahlen oder Zeichenketten, sondern Objekte der anderen drei bereits modellierten Objektarten. Wenn wir nämlich eine Karteikarte schreiben, dann erfinden wir gerade einen neuen Typ, den wir für die Eigenschaften anderer Karteikarten benutzen können.

Somit erstellen wir eine Karteikarte für den Objekttyp *Ausleihe*, wie sie in Abbildung 2.4 zu sehen ist.

2.2 Klassen und Objekte

Wir haben in einem Modellierungsschritt im letzten Abschnitt verschiedene Objektarten identifiziert und ihre Eigenschaften spezifiziert. Dazu haben wir vier Karteikarten geschrieben. Jetzt können wir versuchen, diese Modellierung in Java umzusetzen. In Java beschreibt eine Klasse eine Menge von Objekten gleicher Art. Damit entspricht eine Klasse einer der Karteikarten in unserer Modellierung. Die

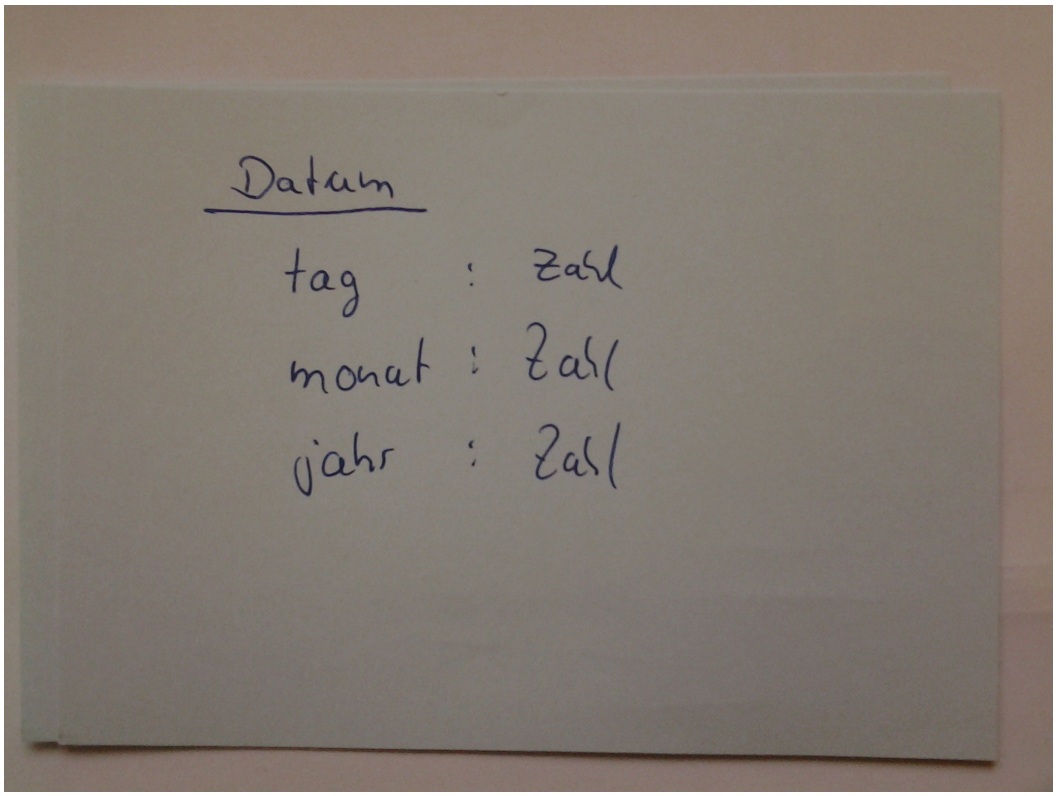

Abbildung 2.3: Modellierung eines Datums.

Klassendefinition ist eine Beschreibung der möglichen Objekte. In ihr ist definiert, was für Daten zu den Objekten gehören. Zusätzlich können wir in einer Klasse noch schreiben, welche Operationen auf diesen Daten angewendet werden können. Klassendefinitionen sind die eigentlichen Programmtexte, die der Programmierer schreibt.

In Java steht genau eine Klassendefinition[a] in genau einer Datei. Die Datei hat dabei den Namen der Klasse mit der Endung `.java+.`

[a] Auch hier werden wir Ausnahmen kennenlernen.

Jede Programmiersprache hat ein paar Wörter, die eine für die Sprache besondere Bedeutung haben. Diese Wörter sind festgelegt und werden als Schlüsselwörter bezeichnet. Die Zahl der Schlüsselwörter ist meist recht klein gehalten.

Zu Beginn einer Klassendefinition steht in Java das Schlüsselwort `class`, gefolgt von dem Namen, den man für die Klasse gewählt hat, deklariert. Der Name der Klasse ist frei wählbar. Wann immer der Programmierer in einer Programmiersprache einen Namen, sei es für eine Klasse, eine Variable oder auch für Felder und Methoden, frei wählen kann, nennt man diese Namen auch *Bezeichner* (engl. *identifier*).

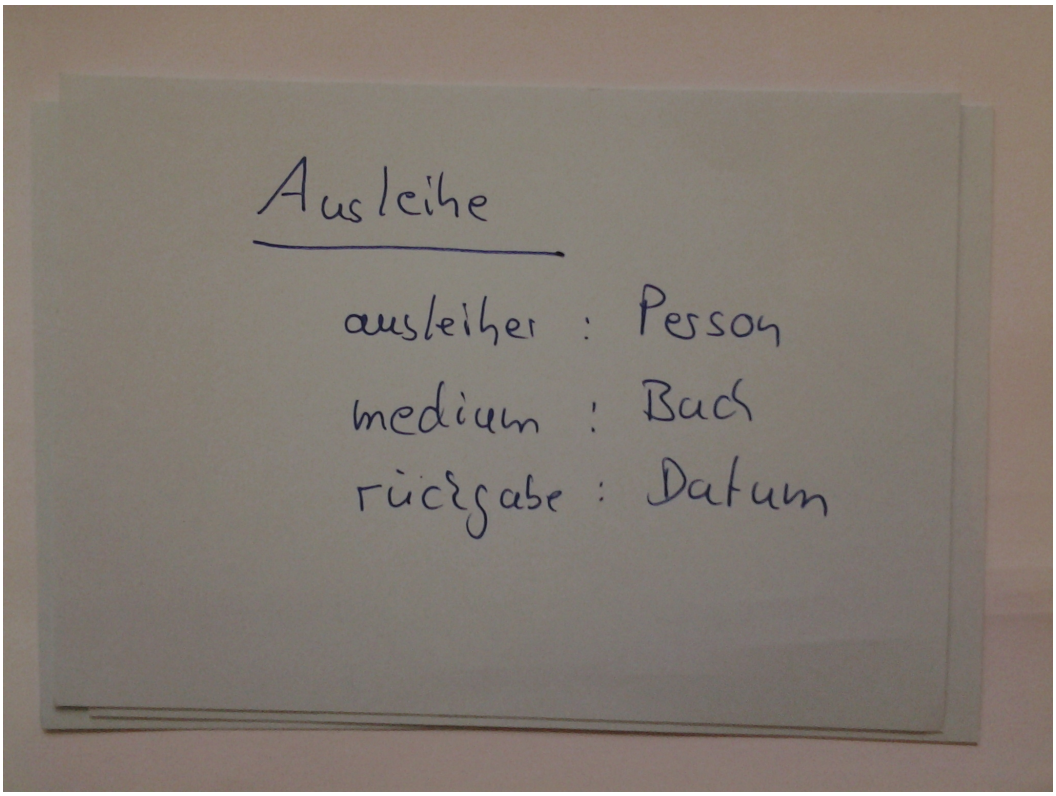

Abbildung 2.4: Modellierung eines Ausleihvorgangs.

Nach dem Bezeichner für den Klassennamen folgt in geschweiften Klammern der Inhalt der Klasse bestehend aus Felddefinitionen und Methodendefinitionen.

Die einfachste Klasse, die in Java denkbar ist, ist eine Klasse ohne Felder oder Methoden:

```
1 class Minimal {
2 }
```

Listing 2.1: Minimal.java

Beachten Sie, dass Groß- und Kleinschreibung in Java relevant ist. Alle Schlüsselwörter wie `class+ werden stets klein geschrieben.` `Klassennamen starten per Konvention immer mit einem Großbuchstaben.`

2.2.1 Felder

Zum Speichern von Daten können Felder (engl. *field*) für eine Klasse definiert werden. In einem Feld können Objekte für eine bestimmte Klasse gespeichert werden.

Bei der Felddeklaration wird angegeben, welche Art von Objekten in einem Feld abgespeichert werden sollen. Die Felder entsprechen dabei genau den Eigenschaften, die wir auf unsere Karteikarten geschrieben haben.

Syntaktisch wird in Java der Klassenname des Typs, von dem Objekte gespeichert werden sollen, den frei zu wählenden Feldnamen vorangestellt. Eine Felddeklaration endet mit einem Semikolon.

Im Folgenden schreiben wir eine Klasse mit drei Feldern:

```
1 class Buch{
2   String autor;
3   String titel;
4   int preisInCent;
5 }
```

Listing 2.2: Buch.java

Für eine Zeichenkette steht in Java die Klasse `String` zur Verfügung. Für ganze Zahlen existiert der Typ `int`.

Die Reihenfolge in der die Felder einer Klasse definiert werden, ist mehr oder weniger irrelevant.

In der Literatur werden Felder auf deutsch auch als Attribute einer Klasse oder auch als Exemplarvariablen bezeichent.

2.2.2 Objekte

Im letzten Abschnitt haben wir eine erste Klasse geschrieben. Sie implementiert die Modellierung von Büchern. Sie hat daher drei Felder, die den Eigenschaften aus der Modellierung entsprechen. Jetzt wollen wir diese Klasse benutzen, um damit konkrete Buchobjekte zu erzeugen, die konkrete Werte für den Titel etc. haben. Hierzu brauchen wir eine Klasse mit einer Hauptmethode, die ausgeführt werden soll. Innerhalb dieser Hauptmethode soll dann ein Buchobjekt erzeugt werden.

Wir beginnen eine entsprechende Klasse zum ersten Testen von Buchobjekten.

```
1 class TestBuch{
2   public static void main(String[] args){
```

Listing 2.3: TestBuch.java

Erzeugen von Objekten mit `new`

Innerhalb dieser Hauptmethoden können wir jetzt konkrete Objekte der Klasse `Buch` erzeugen. Hierzu gibt es in Java das Schlüsselwort `new`. Mit diesem wird angezeigt,

dass ein neues Objekt erzeugt werden soll. Dem Schlüsselwort folgt der Name der Klasse, für die ein neues Objekt zu erzeugen ist. In unserem Fall also `Buch`. Anschließend folgt noch ein rundes Klammernpaar, also indsgesamt `new Buch()`.

Wurde so ein Objekt erzeugt, ist dieses irgendwo abzuspeichern. Hierzu dienen lokale Variablen. Diese haben einen Bezeichner. Über diesen Bezeichner wird die Variable angesprochen. Zusätzlich ist noch durch einen Typnamen anzugeben, welche Art von Daten in dieser Variable gespeichert werden sollen. In unserem Fall ist dieses der Typ `Buch`, denn es sollen Buchobjekte in der Variablen abgelegt werden.

```
3   Buch b1 = new Buch();
```

Listing 2.4: TestBuch.java

Das Gleichheitszeichen in dieser Zeile steht für die sogenannte *Zuweisung* (eng.: *assignment*). Der Variablen `b1` wird hiermit ein konkretes Objekt zugewiesen.

Zugriff auf die Felder von Objekten

Von nun an bezeichnet die Variable `b1` ein konkretes Objekt der Klasse `Buch`. Allerdings haben wir für dieses Objekt noch keine konkreten Werte für die Felder zugewiesen. Der Titel, der Autor und auch der Preis für dieses Buchobjekt wurde noch nicht gesetzt. Auf die Felder eines Objektes kann mit einem Punkt zugegriffen werden. Links von dem Punkt steht ein Objekt, rechts davon ein Feldname. So können wir in unserem Beispiel mit `b1.titel` auf den Titel des Buchobjektes `b1` zugreifen. Um einen Titel zu setzen, kann wieder der Zuweisungsbefehl genutzt werden. Wir können alle drei Felder mit folgenden Zuweisungen mit konkreten Werten belegen:

```
4       b1.titel = "Geschüttelt, nicht gerührt!";
5       b1.autor = "Matthias Oheim, Sven Eric Panitz";
6       b1.preisInCent = 750;
```

Listing 2.5: TestBuch.java

Wie man sieht, lassen sich Zeichenketten in Java direkt in doppelten Anführungszeichen eingeschlossen hinschreiben. Dabei ist zu beachten, das diese so notierten Stringobjekte nicht über ein Zeilenende hinaus gehen dürfen.

Konstante Zahlen lassen sich in Java wie eigentlich in allen Programmiersprachen direkt hinschreiben.

Nach den obigen Zuweisungen hat das Objekt `b1` konkrete Werte. Mit dem Ausgabebefehl, den wir im vorherigen Kapitel vorgestellt haben, lassen sich die einzelnen Daten der Felder eines Objektes auf der Kommandozeile ausgeben:

```
7       System.out.println(b1.autor);
```

Listing 2.6: TestBuch.java

Klassen werden geschrieben, um nicht nur ein Objekt, sondern zumeist sehr viele Objekte einer Klasse zu erzeugen. Somit können wir jetzt ein zweites Buchobjekt erzeugen und diesem konkrete Werte für die Felder zuweisen.

```
8       Buch b2 = new Buch();
9       b2.titel = "Gerüttelt, nicht geschürt!";
10      b2.autor = "Matthias Oheim, Sven Eric Panitz";
11      b2.preisInCent = 980;
```

Listing 2.7: TestBuch.java

Die Objekte einer Klasse sind voneinander unabhängig. Beide Objekte haben ihre eigenen Werte. Davon können wir uns durch die Ausgabe der Titel beider Objekte überzeugen.

```
12      System.out.println(b1.titel);
13      System.out.println(b2.titel);
```

Listing 2.8: TestBuch.java

Soweit unsere erste Klasse, die mit Buchobjekten arbeitet. Es fehlen noch die zwei schließenden geschweiften Klammern. Einmal, um anzuzeigen, dass die Hauptmethode beendet wird und einmal um das Ende des Klassendefinition anzuzeigen.

```
14    }
15  }
```

Listing 2.9: TestBuch.java

Konstruktoren

Wir haben oben gesehen, wie prinzipiell Objekte einer Klasse mit dem `new`-Konstrukt erzeugt werden. In unserem obigen Beispiel würden wir gerne bei der Erzeugung eines Objektes gleich konkrete Werte für die Felder mit angeben, um direkt eine Person mit konkreten Namen erzeugen zu können. Hierzu können Konstruktoren für eine Klasse definiert werden.

```
1   class Person {
2     String vorname;
3     String nachname;
4
5     Person(String derVorname, String derNachname){
```

```
6       vorname  = derVorname;
7       nachname = derNachname;
8     }
9   }
```

Listing 2.10: Person.java

Jetzt lassen sich bei der Erzeugung von Objekten des Typs `Person` konkrete Werte für die Namen übergeben.

Wir erzeugen ein Personenobjekt mit dem für die entsprechende Klasse geschriebenen Konstruktor:

```
1  class TestePerson {
2
3    public static void main(String [] _){
4      Person p = new Person("Nicolo","Paganini");
5      System.out.println(p.vorname);
6    }
7  }
```

Listing 2.11: TestePerson1.java

2.2.3 Objekte der Klasse `String`

Java kommt bereits mit einer großen Anzahl zur Verfügung stehender Standardklassen. Es müssen also nicht alle Klassen neu vom Programmierer definiert werden. Eine sehr häufig benutzte Klasse ist die Klasse `String`. Sie repräsentiert Objekte, die eine Zeichenkette darstellen, also einen Text, wie wir ihn in unserer ersten Modellierung bereits vorausgesetzt haben.

Für die Klasse `String` gibt es eine besondere Art, Objekte zu erzeugen. Ein in Anführungsstrichen eingeschlossener Text erzeugt ein Objekt der Klasse `String`.

Aus zwei Objekten der Stringklasse läßt sich ein neues Objekt erzeugen, indem diese beiden Objekte mit einem Pluszeichen verbunden werden:

```
1  "hallo "+"welt"
```

Hier werden die zwei Stringobjekte `"hallo "` und `"welt"` zum neuen Objekt `"hallo welt"` verknüpft.

Der Plus-Operator hat für Objekte der Stringklasse noch eine besondere Bedeutung und stellt damit eine Besonderheit innerhalb der Programmiersprache Java dar.

```
1  class SimpleStringTests{
2    public static void main(String[] args){
3      String s1 = "hallo";
4      String s2 = "welt";
5      int i = 42;
6      String s3 = s1+" "+s2+": "+42;
7      System.out.println(s2);
8      System.out.println(""+i+1);
9      System.out.println(""+(i+1));
10   }
11 }
```

Listing 2.12: SimpleStringTests.java

2.2.4 Methoden

Methoden [1] sind die Programmteile, die in einer Klasse definiert sind und für jedes Objekt dieser Klasse zur Verfügung stehen. Die Ausführung einer Methode liefert meist ein Ergebnisobjekt. Methoden haben eine Liste von Eingabeparametern. Ein Eingabeparameter ist durch den gewünschten Klassennamen und einen frei wählbaren Parameternamen spezifiziert.

Aufgabe 2 Implementieren Sie die Klassen `Person`, `Buch`, `Datum` und `Ausleihe` entsprechend der Modellierung in diesem Kapitel. Versehen Sie die Klassen mit einem adäquaten Konstruktor und einer Methode `toString`. Schreiben Sie eine Hauptmethode, in der für jede Klasse mindestens ein Objekt erzeugt wird.

Aufgabe 3 Schreiben Sie eine Klasse `Vertex`, die Punkte in einem zweidimensionalen Raum repräsentiert. Es soll hierzu eine x- und eine y-Koordinate geben. Diese haben als Typ eine Fließkommazahl des eingebauten Javatyps `double`.

a) Versehen Sie die Klasse mit einem adäquaten Konstruktor und einer Methode `toString`.

b) Schreiben Sie in der Klasse eine Methode: `void move(Vertex v)`, die den Punkt um die x- und y-Werte des Parameter `v` verschiebt.

c) Schreiben Sie in der Klasse eine Methode: `void moveTo(Vertex v)`, die den Punkt auf die x- und y-Werte des Parameter `v` setzt.

d) Testen Sie Ihre Klasse mit ein paar Aufrufen in einer Hauptmethode.

Aufgabe 4 Schreiben Sie eine Klasse `GeometricObject`. Diese habe ein Feld `corner` des Typs `Vertex`, ein Feld `width` des Typs `double` und eine Feld

[1] Der Ausdruck für *Methoden* kommt speziell aus der objektorientierten Programmierung. In der imperativen Programmierung spricht man von *Prozeduren*, die funktionale Programmierung von *Funktionen*. Weitere Begriffe, die Ähnliches beschreiben, sind *Unterprogramme* und *Subroutinen*.

`height` ebenfalls des Typs `double`. Versshen Sie die Klasse mit einem geeigneten Konstruktor und einer Methode `toString`.

Methodendeklaration

In Java wird eine Methode deklariert durch: den Rückgabetyp, den Namen der Methode, der in Klammern eingeschlossenen durch Kommas getrennten Parameterliste und den in geschweiften Klammern eingeschlossenen Programmrumpf. Im Programmrumpf wird mit dem Schlüsselwort `return` angegeben, welches Ergebnisobjekt die Methode liefert.

Als Beispiel definieren wir eine Klasse, in der es eine Methode `addString` gibt, die den Ergebnistyp `String` und zwei Parameter vom Typ `String` hat:

```
1 class StringUtilMethod {
2   String addStrings(String leftText, String rightText){
3     return leftText+rightText;
4   }
5 }
```

Listing 2.13: StringUtilMethod.java

Methoden und Parameternamen werden per Konvention immer klein geschrieben.

Zugriff auf Felder im Methodenrumpf

In einer Methode stehen die Felder der Klasse zur Verfügung[2].

Wir können mit den bisherigen Mitteln eine kleine Klasse definieren, die es erlaubt, Personen zu repräsentieren, so dass die Objekte dieser Klasse eine Methode haben, um den vollen Namen der Person anzugeben:

```
1 class PersonExample1 {
2   String vorname;
3   String nachname;
4
5   String getFullName(){
6     return (vorname+" "+nachname);
7   }
8 }
```

Listing 2.14: PersonExample1.java

[2]Das ist wiederum nicht die volle Wahrheit, wie in Kürze zu sehen sein wird.

Methoden ohne Rückgabewert

Es lassen sich auch Methoden schreiben, die keinen eigentlichen Wert berechnen, den sie als Ergebnis zurückgeben. Solche Methoden haben keinen Rückgabetyp. In Java wird dieses gekennzeichnet, indem das Schlüsselwort `void` statt eines Typnamens in der Deklaration steht. Solche Methoden haben keine `return`-Anweisung.

Folgende kleine Beispielklasse enthält zwei Methoden zum Setzen neuer Werte für ihre Felder:

```
1  class PersonExample2 {
2    String vorname;
3    String nachname;
4
5    void setVorname(String newName){
6      vorname = newName;
7    }
8    void setNachname(String newName){
9      nachname = newName;
10   }
11 }
```

Listing 2.15: PersonExample2.java

Obige Methoden weisen konkrete Objekte den Feldern des Objektes zu.

Kapitel 3

Imperative und funktionale Konzepte

Im letzten Abschnitt wurde ein erster Einstieg in die objektorientierte Programmierung gegeben. Wie zu sehen war, ermöglicht die objektorientierte Programmierung, das zu lösende Problem in logische Untereinheiten zu unterteilen, die direkt mit den Teilen der zu modellierenden Problemwelt korrespondieren.

Die Methodenrümpfe, die die eigentlichen Befehle enthalten, in denen etwas berechnet werden soll, waren bisher recht kurz. In diesem Kapitel werden wir Konstrukte kennenlernen, die es ermöglichen, in den Methodenrümpfen komplexe Berechnungen vorzunehmen. Die in diesem Abschnitt vorgestellten Konstrukte sind herkömmliche Konstrukte der imperativen Programmierung und in ähnlicher Weise auch in Programmiersprachen wie C zu finden.[1]

3.1 Primitive Typen

Bisher haben wir noch überhaupt keine Berechnungen im klassischen Sinne als das Rechnen mit Zahlen kennengelernt. Java stellt Typen zur Repräsentation von Zahlen zur Verfügung. Leider sind diese Typen keine Klassen; d.h. insbesondere, dass auf diesen Typen keine Felder und Methoden existieren, auf die mit einem Punkt zugegriffen werden kann.

Die im Folgenden vorgestellten Typen nennt man primitive Typen. Sie sind fest von Java vorgegeben. Im Gegensatz zu Klassen, die der Programmierer selbst definieren kann, können keine neuen primitiven Typen definiert werden. Um primitive Typnamen von Klassennamen leicht textuell unterscheiden zu können, sind sie in Kleinschreibung definiert worden.

Ansonsten werden primitive Typen genauso behandelt wie Klassen. Felder können primitive Typen als Typ haben und ebenso können Parametertypen und Rückgabetypen von Methoden primitive Typen sein.

[1] Gerade in diesem Bereich wollten die Entwickler von Java einen leichten Umstieg von der C-Programmierung nach Java ermöglichen. Leider hat Java in dieser Hinsicht auch ein C-Erbe und ist nicht in allen Punkte so sauber entworfen, wie es ohne diese Designvorgabe wäre.

Um Daten der primitiven Typen aufschreiben zu können, gibt es jeweils Literale für die Werte dieser Typen.

3.1.1 Zahlenmengen in der Mathematik

In der Mathematik sind wir gewohnt, mit verschiedenen Mengen von Zahlen zu arbeiten:

- **natürliche Zahlen** N: Eine induktiv definierbare Menge mit einer kleinsten Zahl, so dass es für jede Zahl eine eindeutige Nachfolgerzahl gibt.
- **ganze Zahlen** Z: Die natürlichen Zahlen erweitert um die mit einem negativen Vorzeichen behafteten Zahlen, die sich ergeben, wenn man eine größere Zahl von einer natürlichen Zahl abzieht.
- **rationale Zahlen** Q: Die ganzen Zahlen erweitert um Brüche, die sich ergeben, wenn man eine Zahl durch eine Zahl teilt, von der sie kein Vielfaches ist.
- **reelle Zahlen** R: Die ganzen Zahlen erweitert um irrationale Zahlen, die sich z.B. aus der Quadratwurzel von Zahlen ergeben, die nicht das Quadrat einer rationalen Zahl sind.
- **komplexe Zahlen** C: Die reellen Zahlen erweitert um imaginäre Zahlen, wie sie benötigt werden, um einen Wurzelwert für negative Zahlen darzustellen.

Es gilt folgende Mengeninklusion zwischen diesen Mengen:

$$N \subset Z \subset Q \subset R \subset C$$

Da bereits N nicht endlich ist, ist keine dieser Mengen endlich.

3.1.2 Zahlenmengen im Rechner

Da wir nur von einer endlich großen Speicherkapazität ausgehen können, lassen sich für keine der aus der Mathematik bekannten Zahlenmengen alle Werte in einem Rechner darstellen. Wir können also schon einmal nur Teilmengen der Zahlenmengen darstellen.

Von der Hardwareseite stellt sich heute zumeist die folgende Situation dar: Der Computer hat einen linearen Speicher, der in Speicheradressen unterteilt ist. Eine Speicheradresse bezeichnet einen Bereich von 32 Bit. Wir bezeichnen diese als ein Wort. Die Einheit von 8 Bit wird als Byte bezeichnet [2]. Heutige Rechner verwalten also in der Regel Dateneinheiten von 32 Bit. Hieraus ergibt sich die Kardinalität

[2] ein anderes selten gebrauchtes Wort aus dem Französischen ist: Oktett

der Zahlenmengen, mit denen ein Rechner als primitive Typen rechnen kann. Soll mit größeren Zahlenmengen gerechnet werden, so muss hierzu eine Softwarelösung genutzt werden.

Ganzzahlentypen in Java

Java 4 Typen zur Darstellung ganzer Zahlen, die sich lediglich in der Anzahl der Ziffern unterscheiden. Die Zahlen werden intern als Zweierkomplement dargestellt.

In der Programmiersprache Java sind die konkreten Wertebereiche für die einzelnen primitiven Typen in der Spezifikation festgelegt. In anderen Programmiersprachen wie z.B. C ist dies nicht der Fall. Hier hängt es vom Compiler und dem konkreten Rechner ab, welchen Wertebereich die entsprechenden Typen haben.

Es gibt Programmiersprachen wie z.B. Haskell, in denen es einen Typ gibt, der potentiell ganze Zahlen von beliebiger Größe darstellen kann.

Starten Sie folgendes Javaprogramm:

```
1 class TestInteger {
2   public static void main(String [] _){
3     System.out.println(2147483647+1);
4     System.out.println(-2147483648-1);
5   }
6 }
```

Listing 3.1: TestInteger.java

Erklären Sie die Ausgabe.

Fließkommazahlen in Java

Eine Alternative zu der Festkommadarstellung von Zahlen ist die Fließkommadarstellung. Während die Festkommadarstellung einen Zahlenbereich der rationalen Zahlen in einem festen Intervall durch diskrete, äquidistant verteilte Werte darstellen kann, sind die diskreten Werte in der Fließkommadarstellung nicht gleich verteilt.

In der Fließkommadarstellung wird eine Zahl durch zwei Zahlen charakterisiert und ist bezüglich einer Basis b :

- die Mantisse für die darstellbaren Ziffern. Die Mantisse charakterisiert die Genauigkeit der Fließkommazahl.
- der Exponent, der angibt, wie weit die Mantisse hinter bzw. vor dem Komma liegt.

Aus Mantisse m , Basis b und Exponent exp ergibt sich die dargestellte Zahl durch folgende Formel:

$$z = m * b^{exp}$$

Damit lassen sich mit Fließkommazahlen sehr große und sehr kleine Zahlen darstellen. Je größer jedoch die Zahlen werden, desto weiter liegen sie von der nächsten Zahl entfernt.

Für die Fließkommadarstellung gibt es in Java zwei Zahlentypen, die nach der Spezifikation des IEEE 754-1985 gebildet werden:

- **float**: 32 Bit Fließkommazahl nach IEEE 754. Kleinste positive Zahl: 2^{-149}. Größte positive Zahl: $(1-2^{-24}) * 2^{128}$
- **double**: 64 Bit Fließkommazahl nach IEEE 754. Kleinste positive Zahl: 2^{-1074}. Größte positive Zahl: $(1 - 2^{-53}) * 2^{1024}$

Im Format für `double` steht das erste Bit für das Vorzeichen, die nächsten 11 Bit markieren den Exponenten und die restlichen 52 Bit kodieren die Mantisse.

Im Format für `float` steht das erste Bit für das Vorzeichen, die nächsten 8 Bit markieren den Exponenten und die restlichen 23 Bit kodieren die Mantisse.

Bestimmte Bitmuster charakterisieren einen Wert für negative und positive unbeschränkte Werte (unendlich) sowie Zahlen, Bitmuster, die charakterisieren, dass es sich nicht mehr um eine Zahl handelt.

Der folgende Test zeigt, dass bei einer Addition von zwei Fließkommazahlen die kleinere Zahl das Nachsehen hat:

```
1  class DoubleTest{
2    public static void main(String [] args){
3      double x = 325e200;
4      double y = 325e-200;
5      System.out.println(x);
6      System.out.println(y);
7      System.out.println(x+y);
8      System.out.println(x+100000);
9    }
10 }
```

Listing 3.2: DoubleTest.java

Wie man an der Ausgabe erkennen kann: selbst die Addition der Zahl `100000` bewirkt keine Veränderung auf einer großen Fließkommazahl:

```
sep@linux:~/fh/prog3/examples/src> java DoubleTest
3.25E202
3.25E-198
3.25E202
3.25E202
sep@linux:~/fh/prog3/examples/src>
```

Das folgende kleine Beispiel zeigt, inwieweit und für den Benutzer oft auf überraschende Weise die Fließkommadarstellung zu Rundungen führt:

```
1  class Rounded {
2    public static void main(String [] args){
3      System.out.println(8f);
4      System.out.println(88f);
5      System.out.println(8888f);
6      System.out.println(88888f);
7      System.out.println(888888f);
8      System.out.println(8888888f);
9      System.out.println(88888888f);
10     System.out.println(888888888f);
11     System.out.println(8888888888f);
12     System.out.println(88888888888f);
13     System.out.println(888888888888f);
14     System.out.println(8888888888888f);
15
16     System.out.println(1f+1000000000000f-1000000000000f);
17   }
18 }
```

Listing 3.3: Rounded.java

Das Programm hat die folgende Ausgabe. Insbesondere in der letzten Zeile fällt auf, dass Addition und anschließende Subtraktion ein und derselben Zahl nicht die Identität ist. Für Fließkommazahlen gilt nicht: $x + y - y = x$.

```
sep@linux:~/fh/prog3/examples/src> java Rounded
8.0
88.0
8888.0
88888.0
888888.0
8888888.0
8.8888888E7
8.888889E8
8.8888893E9
8.8888885E10
8.8888889E11
8.8888889E12
0.0
sep@linux:~/fh/prog3/examples/src>
```

3.2 Ausdrücke

Wir haben jetzt gesehen, was Java uns für Typen zur Darstellung von Zahlen zur Verfügung stellt. Jetzt wollen wir mit diesen Zahlen nach Möglichkeit auch noch

rechnen können. Hierzu stellt Java eine feste Anzahl von Operatoren wie `*, -, /` etc. zur Verfügung. Prinzipell gibt es in der Informatik für Operatoren drei mögliche Schreibweisen:

- **Präfix:** Der Operator wird vor den Operanden geschrieben, also z.B. `(* 2 21)`. Im ursprünglichen Lisp gab es die Prefixnotation für Operatoren.
- **Postfix:** Der Operator folgt den Operanden, also z.B. `(21 2 *)`. Forth und Postscript sind Beispiele von Sprachen mit Postfixnotation.
- **Infix:** Der Operator steht zwischen den Operanden. Dieses ist die gängige Schreibweise in der Mathematik und für das Auge die gewohnteste. Aus diesem Grunde bedient sich Java der Infixnotation: `42 * 2`.

Aufgabe 1 Nehmen Sie die Klasse `Datum` aus dem letzten Übungsblatt und ergänzen Sie diese um folgende Eigenschaften:

a) Schreiben Sie eine Methode:

`boolean isEarlierThan(Datum that)`,

die testet ob das Datumsobjekt im Kalender vor einem anderen Datum liegt.

b) Schreiben Sie eine Methode:

`boolean isLaterThan(Datum that)`,

die testet ob das Datumsobjekt im Kalender nach einem anderen Datum liegt.

c) Schreiben Sie eine Methode:

`boolean sameDay(Datum that)`,

die testet, ob zwei Datumsobjekte den gleichen Tag bezeichnen.

d) Schreiben Sie eine Methode:

`boolean isLeapYear()`,

die testet, ob das Datum in einem Schaltjahr liegt.

e) Schreiben Sie eine Methode:

`int getAbsoluteDaysInYear()`,

die zurück gibt, wie viel Tage das Jahr des Datums hat.

f) Schreiben Sie eine Methode:

`int getDaysInMonth()`,

die zurück gibt, wie viel Tage der Monat des Datums hat.

Aufgabe 2 Sie sollen in dieser Aufgabe berechnen, was für einen Wochentag, ein bestimmtes Datum hat. Ergänzen Sie die Klasse `Datum` um Methoden, die helfen den Wochentag zu berechnen. Ein Algorithmus zur Wochentagsberechnung finden Sie z.B. auf: . Wikipedia (`http://de.wikipedia.org/wiki/Wochentagsberechnung`)

Gehen Sie schrittweise vor und schreiben Methoden zur Berechnung der Tagesziffer, der Monatsziffer, der Jahresziffer, der Jahrhundertziffer und der Schaltjahrskorrektur, wie in dem Algorithmus spezifiziert. Dann können Sie mit folgenden Methoden den Wochentag berechnen:

```
int wochentag(){
   return (tagesziffer()+monatsziffer()+jahresziffer()
          +jahrhundertZiffer()+schaltjahrKorrektur())
           % 7;
  }

  String wochenTagAlsString(){
    int wtag = wochentag();
     switch (wtag){
       case 0: return "Sonntag";
       case 1: return "Montag";
       case 2: return "Dienstag";
       case 3: return "Mittwoch";
       case 4: return "Donnerstag";
       case 5: return "Freitag";
       default: return "Sonnabend";
     }
  }
```

Schreiben Sie Testaufrufe für die Wochentagsberechnung.

Aufgabe 3 Ergänzen Sie die Klasse Datum nun noch um eine Methode, die für ein bestimmtes Datumsobjekt berechnet, an welchen Datum in dem entsprechenden Jahr Ostern liegt:

`Datum getEasterDay()`

Es soll für das Jahr des Datums der Ostersonntag errechnet werden. Hierzu können Sie die ergänzte Osterformel benutzen, wie auf Wikipedia beschrieben (`de.wikipedia.org/wiki/Osterformel`) .

3.2.1 Arithmetische Operatoren

Java stellt für Zahlen die vier Grundrechenarten zur Verfügung.

Bei der Infixnotation gelten für die vier Grundrechenarten die üblichen Regeln der Bindung, nämlich Punktrechnung vor Strichrechnung. Möchte man diese Regel durchbrechen, so sind Unterausdrücke in Klammern zu setzen. Folgende kleine Klasse demonstriert den Unterschied:

```
1 class PunktVorStrich{
2   public static void main(String [] args){
3     System.out.println(17 + 4 * 2);
4     System.out.println((17 + 4) * 2);
5   }
6 }
```

Listing 3.4: PunktVorStrich.java

Wir können nun also Methoden schreiben, die Rechnungen vornehmen. In der folgenden Klasse definieren wir z.B. eine Methode zum Berechnen der Quadratzahl der Eingabe:

```
1 class Square{
2   static int square(int i){
3     return i*i;
4   }
5 }
```

Listing 3.5: Square.java

Für die Division auf Ganzzahlen gibt es zwei Operatoren. Den eigentlichen Divisionsoperator `/` und den Operator `%`, der den ganzzahligen Rest einer Division anzeigt, der sogenannte Modulooperator:

```
1 class DivTest{
2   public static void main(String [] args){
3     System.out.println(10/3);
4     System.out.println(10%3);
5     System.out.println(-10/3);
6     System.out.println(-10%3);
7   }
8 }
```

Listing 3.6: DivTest.java

3.2.2 Vergleichsoperatoren

Obige Operatoren rechnen jeweils auf zwei Zahlen und ergeben wieder eine Zahl als Ergebnis. Vergleichsoperatoren vergleichen zwei Zahlen und geben einen bool'schen Wert, der angibt, ob der Vergleich wahr oder falsch ist. Java stellt die folgenden Vergleichsoperatoren zur Verfügung: `<`, `<=`, `>`, `>=`, `!=`, `==`. Für die Gleichheit ist in Java das doppelte Gleichheitszeichen `==` zu schreiben, denn das einfache Gleichheitszeichen ist bereits für den Zuweisungsbefehl vergeben. Die Ungleichheit wird mit `!=` bezeichnet.

Folgende Tests demonstrieren die Benutzung der Vergleichsoperatoren:

```
class Vergleich{

  public static void main(String[] args){
    System.out.println(1+1 < 42);
    System.out.println(1+1 <= 42);
    System.out.println(1+1 >  42);
    System.out.println(1+1 >= 42);
    System.out.println(1+1 == 42);
    System.out.println(1+1 != 42);
  }
}
```

Listing 3.7: Vergleich.java

3.2.3 Logische Operatoren

In der bool'schen Logik gibt es eine ganze Reihe von binären Operatoren für logische Ausdrücke. Für zwei davon stellt Java auch Operatoren bereit: `&&` für das logische Und $\wedge$ und `||` für das logische Oder $\vee$.

Zusätzlich kennt Java noch den unären Operator der logischen Negation $\neg$. Er wird in Java mit `!` bezeichnet.

Wie man im folgenden Test sehen kann, gibt es auch unter den bool'schen Operatoren eine Bindungspräzedenz, ähnlich wie bei der Regel Punktrechnung vor Strichrechnung. Der Operator `&&` bindet stärker als der Operator `||`:

```
class TestboolOperator{
  public static void main(String [] args){
    System.out.println(true && false);
    System.out.println(true || false);
    System.out.println(!true || false);
    System.out.println(true || true && false);
  }
}
```

Listing 3.8: TestboolOperator.java

In der formalen Logik kennt man noch weitere Operatoren, z.B. die Implikation $\rightarrow$. Diese Operatoren lassen sich aber durch die in Java zur Verfügung stehenden Operatoren ausdrücken. $A \rightarrow B$ entspricht $\neg A \vee B$. Wir können somit eine Methode schreiben, die die logische Implikation testet:

```
1 class TestboolOperator2{
2   static boolean implication(boolean a, boolean b){
3     return !a || b;
4   }
5
6   public static void main(String [] args){
7     System.out.println(implication(true, false));
8   }
9 }
```

Listing 3.9: TestboolOperator2.java

3.2.4 Der Bedingungsoperator

Java kennt auch einen Operator mit drei Operanden. Er besteht aus zwei einzelnen Zeichen, die als Trenner zwischen den Operanden stehen. Zunächst kommt der erste Operand, dann das Zeichen `?`, dann der zweite Operand, gefolgt vom Zeichen `:`, dem der dritte Operand folgt. Schematisch sehen die Ausdrücke dieses Operators wie folgt aus:

cond ? alt$_1$ *: alt*$_2$

Das Ergebnis dieses Operators wird wie folgt berechnet: Der erste Operand wird zu einem Wahrheitswert ausgewertet. Wenn dieser *wahr* ist, so wird der zweite Operand als Ergebnis ausgewertet, wenn er *falsch* ist, wird der dritte Operand als Ergebnis ausgewertet.

```
1 class SoOderSo{
2   public static void  main(String[] args){
3     System.out.println((17+4)*2==42?"tatsaechlich gleich":"
    unterschiedlich");
4   }
5 }
```

Listing 3.10: SoOderSo.java

Der Bedingungsoperator ist unser erstes Konstrukt, um Verzweigungen auszudrücken. Da der Bedingungsoperator auch einen Wert errechnet, können wir diesen benutzen, um mit ihm weiterzurechenen. Der Bedingungsoperator kann also tatsächlich ineinandergesteckt werden:

```
class Signum1{
  public static void  main(String[] args){
    System.out.println("signum(42) = "+((42>0)?1:((42<0)?-1:0)));
  }
}
```

Listing 3.11: Signum1.java

Hier wird zunächst geschaut, ob die Zahl 42 größer als 0 ist. Ist dieses der Fall wird die Zahl 1 ausgegeben, ansonsten wird weitergeschaut, ob die Zahl 42 kleiner 1 ist. Hier wird im Erfolgsfall die Zahl -1 ausgegeben. Wenn beides nicht der Fall war, wird die Zahl 0 ausgegeben.

Zugegebener Maßen ist dieser Ausdruck schon schwer zu lesen. Wir werden später bessere Konstrukte kennenlernen, um verschiedene Fälle zu unterscheiden.

3.3 Anweisungen

3.3.1 Fallunterscheidungen

Bedingungsabfrage mit `if`

Ein häufig benötigtes Konstrukt ist, dass ein Programm abhängig von einer bool'schen Bedingung sich verschieden verhält. Hierzu stellt Java die **if**-Bedingung zur Verfügung. Dem Schlüsselwort `if` folgt in Klammern eine bool'sche Bedingung, anschließend kommen in geschweiften Klammern die Befehle, die auszuführen sind, wenn die Bedingung wahr ist. Anschließend kann optional das Schlüsselwort `else` folgen mit den Befehlen, die andernfalls auszuführen sind:

```
class FirstIf {

  static void firstIf(boolean bedingung){
    if (bedingung) {
      System.out.println("Bedingung ist wahr");
    } else {
      System.out.println("Bedingung ist falsch");
    }
  }

  public static void main(String [] args){
     firstIf(true || false);
  }
}
```

Listing 3.12: FirstIf.java

Das `if`-Konstrukt erlaubt es uns also, Fallunterscheidungen zu treffen. Wenn in den Alternativen nur ein Befehl steht, so können die geschweiften Klammern auch fortgelassen werden. Unser Beispiel läßt sich also auch schreiben als:

```
1  class FirstIf2 {
2
3    static void firstIf(boolean bedingung){
4      if (bedingung) System.out.println("Bedingung ist wahr");
5      else System.out.println("Bedingung ist falsch");
6    }
7
8    public static void main(String [] args){
9       firstIf(true || false);
10   }
11
12 }
```

Listing 3.13: FirstIf2.java

Eine Folge von mehreren `if`-Konstrukten läßt sich auch direkt hintereinanderschreiben, so dass eine Kette von `if`- und `else`-Klauseln entsteht:

```
1  class ElseIf {
2
3    static String lessOrEq(int i,int j){
4      if (i<10)  return "i kleiner zehn";
5      else if (i>10)  return "i größer zehn";
6      else if (j>10)  return "j größer zehn";
7      else if (j<10)  return "j kleiner zehn";
8      else return "j=i=10";
9    }
10
11   public static void main(String [] args){
12      System.out.println(lessOrEq(10,9));
13   }
14 }
```

Listing 3.14: ElseIf.java

Wenn zuviele `if`-Bedingungen in einem Programm einander folgen und ineinander verschachtelt sind, dann wird das Programm schnell unübersichtlich. Man spricht auch von *Spaghetti-code*. In der Regel empfiehlt es sich, in solchen Fällen noch einmal über das Design nachzudenken, ob die abgefragten Bedingungen sich nicht durch verschiedene Klassen mit eigenen Methoden darstellen lassen.

Fallunterscheidungen mit `switch`

Aus C erbt Java eine sehr spezielle zusammengesetzte Anweisung, die **switch**-Anweisung. Es ist eine Anweisung für eine Fallunterscheidung mit mehreren Fällen,

die `switch`-Anweisung. Die Idee dieser Anweisung ist, eine Kette von mehreren `if-then`-Anweisungen zu vermeiden. Leider ist die `switch`-Anweisung in seiner Anwendungsbereite recht begrenzt und in Form und Semantik ziemlich veraltet.

Schematisch hat die `switch`-Anweisung die folgende Form:

switch (*expr*){
case *const*: *stats*
...
case *const*: *stats*
default: *stats*
case *const*: *stats*
...
case *const*: *stats*
}

Dem Schlüsselwort `switch` folgt ein Ausdruck, nach dessen Wert eine Fallunterscheidung getroffen werden soll. In geschweiften Klammern folgen die verschiedenen Fälle. Ein Fall beginnt mit dem Schüsselwort `case` gefolgt von einer Konstante. Diese Konstante ist von einem ganzzahligen Typ und darf kein Ausdruck sein, der erst während der Laufzeit berechnet wird. Es muss hier eine Zahl stehen. Die Konstante muss während der Übersetzungszeit des Programms feststehen. Der Konstante folgt ein Doppelpunkt, dem dann die Anweisungen für diesen Fall folgen. Ein besonderer Fall ist der `default`-Fall. Dieses ist der Standardfall. Er wird immer ausgeführt, egal was für einen Wert der Ausdruck nach dem die `switch`-Anweisung unterscheidet hat.

Ein kleines Beispiel soll die operationale Semantik dieser Anweisung verdeutlichen.

```
1  class Switch {
2
3    public static void main (String [] args){
4      switch (4*new Integer(args[0]).intValue()){
5        case 42 : System.out.println(42);
6        case 52 : System.out.println(52);
7        case 32 : System.out.println(32);
8        case 22 : System.out.println(22);
9        case 12 : System.out.println(12);
10       default : System.out.println("default");
11     }
12   }
13 }
```

Listing 3.15: Switch.java

Starten wir das Programm mit dem Wert `13`, so dass der Ausdruck, nach dem wir die Fallunterscheidung durchführen zu `52` auswertet, so bekommen wir folgende Ausgabe:

```
sep@swe10:~/fh/internal/beispiele> java Switch 13
52
32
22
12
default
sep@swe10:~/fh/internal/beispiele>
```

Wie man sieht, springt die `switch`-Anweisung zum Fall für den Wert `52`, führt aber nicht nur dessen Anweisungen aus, sondern alle folgenden Anweisungen.

Das oben beobachtete Verhalten ist verwirrend. Zumeist will man in einer Fallunterscheidung, dass nur die entsprechenden Anweisung für den vorliegenden Fall ausgeführt werden und nicht auch für alle folgenden Fälle. Um dieses zu erreichen, gibt es die `break`-Anweisung, wie wir sie auch schon von den Schleifenanweisungen kennen. Endet man jeden Fall mit der `break`-Anweisung, dann erhälz man das meistens erwünschte Verhalten.

Das obige Beispiel läßt sich durch Hinzufügen der `break`-Anweisung so ändern, dass immer nur ein Fall ausgeführt wird.

```
1  class Switch2 {
2
3    public static void main (String [] args){
4      switch (4*new Integer(args[0]).intValue()){
5        case 42 : System.out.println(42);break;
6        case 52 : System.out.println(52);break;
7        case 32 : System.out.println(32);break;
8        case 22 : System.out.println(22);break;
9        case 12 : System.out.println(12);break;
10       default : System.out.println("default");
11     }
12   }
13 }
```

An der Ausgabe sehen wir, dass zu einem Fall gesprungen wird und am Ende dieses Falls die Anweisung verlassen wird.

```
sep@swe10:~/fh/internal/beispiele> java Switch2 13
52
sep@swe10:~/fh/internal/beispiele>
```

3.3.2 Iteration

Schleifen mit `while`

Vorgeprüfte Schleifen Die vorgeprüften Schleifen haben folgendes Schema in Java:

$$\text{while } (\textit{pred})\{\textit{body}\}$$

pred ist hierbei ein Ausdruck, der zu einem bool'schen Wert auswertet. *body* ist eine Folge von Befehlen. Java arbeitet die vorgeprüfte Schleife ab, indem erst die Bedingung *pred* ausgewertet wird. Ist das Ergebnis `true`, dann wird der Rumpf *(body)* der Schleife durchlaufen. Anschließend wird wieder die Bedingung geprüft. Dieses wiederholt sich so lange, bis die Bedingung zu `false` auswertet.

Ein einfaches Beispiel ist eine Schleife, deren Bedingung nie zu `false` ausgewertet wird. Eine solche Schleife wird unendlich oft durchlaufen.

```
1 class ForeverYoung{
2   public static void main(String[] args){
3     while (true){
4       System.out.println("young");
5     }
6   }
7 }
```

Listing 3.16: ForeverYoung.java

Wir können eine Variable einführen, die bei jedem Schleifendurchlauf um eins erhöht wird. Somit zählt die Variable, wie oft die Schleife durchlaufen wird.

```
1 class CountForeverYoung{
2   public static void main(String[] args){
3     int i = 0;
4     while (true){
5       System.out.println("young: "+i);
6       i = i + 1;
7     }
8   }
9 }
```

Listing 3.17: CountForeverYoung.java

Diese Variable kann nun dazu genutzt werden, die Schleifen nach einer bestimmten Anzahl von Durchläufen zu beenden, in dem die Schleifenbedingung dieses ausdrückt.

```
class NotForeverYoung{
  public static void main(String[] args){
    int i = 0;
    while (i < 20){
      System.out.println("young: "+i);
      i = i + 1;
    }
  }
}
```

Listing 3.18: NotForeverYoung.java

Wie man an den Beispielen sieht, gibt es oft eine lokale Variable, die zur Steuerung der Schleife benutzt wird. Dieses verändert innerhalb des Schleifenrumpfes seinen Wert. Abhängig von diesem Wert wird die Schleifenbedingung beim nächsten Bedingungstest wieder wahr oder falsch. Eine Variable, von der die Schleifenbedingung abhängt und die im Schleifenrumpf verändert wird, bezeichnet man als Schleifenvariable.

Nun kann man mit einer Schleife nach und nach ein Ergebnis in Abhängigkeit von der Schleifenvariable errechnen. Folgende Methode errechnet die Summe aller Zahlen in einem bestimmten Zahlenbereich.

```
class Summe2 {
  public static int summe(int n){

    int erg = 0 ;              // Ergebnisvariable.
    int j   = n;              // Schleifenvariable.

    while (j>0){               // j läuft von n bis 1.
      erg = erg + j;           // akkumuliere das Ergebnis.
      j = j-1;                 // verringere Laufzähler.
    }

    return erg;
  }
}
```

Listing 3.19: Summe2.java

Nachgeprüfte Schleifen In der zweiten Variante der `while`-Schleife steht die Schleifenbedingung syntaktisch nach dem Schleifenrumpf:

$$\text{do } \{body\} \text{ while } (pred)$$

Bei der Abarbeitung einer solchen Schleife wird entsprechend der Notation, die Bedingung erst nach der Ausführung des Schleifenrumpfes geprüft. Am Ende wird

also geprüft, ob die Schleife ein weiteres Mal zu durchlaufen ist. Das impliziert insbesondere, dass der Rumpf mindestens einmal durchlaufen wird.

```
class DoTest {
  public static void main(String [] args){
    int i = 0;
    do {
      System.out.println(i);
      i = i+1;
    } while (i < 10);
  }
}
```

Listing 3.20: DoTest.java

Man kann sich leicht davon vergewissern, dass die nachgeprüfte Schleife mindestens einmal durchlaufen[3] wird:

```
class VorUndNach {

  public static void main(String [] args){

      while (falsch())
         {System.out.println("vorgeprüfte Schleife");};

      do {System.out.println("nachgeprüfte Schleife");}
      while (false);
  }

  public static boolean falsch(){return false;}
}
```

Listing 3.21: VorUndNach.java

Schleifen mit `for`

Das syntaktisch aufwendigste Schleifenkonstrukt in Java ist die *for*-Schleife.

Wer sich die obigen Schleifen anschaut, sieht, dass sie an drei verschiedenen Stellen im Programmtext Code haben, der kontrolliert, wie oft die Schleife zu durchlaufen ist. Oft legen wir ein spezielles Feld an, dessen Wert die Schleife kontrollieren soll. Dann gibt es im Schleifenrumpf einen Zuweisungsbefehl, der den Wert dieses Feldes verändert. Schließlich wird der Wert dieses Feldes in der Schleifenbedingung abgefragt.

Die Idee der *for*-Schleife ist, diesen Code, der kontrolliert, wie oft die Schleife durchlaufen werden soll, im Kopf der Schleife zu bündeln. Solche Daten sind oft Zähler

[3] Der Javaübersetzer macht kleine Prüfungen auf konstanten Werten, ob Schleifen jeweils durchlaufen werden oder nicht terminieren. Deshalb brauchen wir die Hilfsmethode `falsch()`.

vom Typ `int`, die bis zu einem bestimmten Wert herunter oder hoch gezählt werden. Später werden wir noch die Standardklasse `Iterator` kennenlernen, die benutzt wird, um durch Listenelemente durchzuiterieren.

Eine *for*-Schleife hat im Kopf

- eine Initialisierung der relevanten Schleifensteuerungsvariablen *(init)*,
- ein Prädikat als Schleifenbedingung *(pred)*
- und einen Befehl, der die Schleifensteuerungsvariable weiterschaltet *(step)*.

$$\text{for } (init,\ pred,\ step)\{body\}$$

Entsprechend sieht unsere jeweilige erste Schleife (die Ausgabe der Zahlen von 0 bis 9) in der `for`-Schleifenversion wie folgt aus:

```
1 class ForTest {
2   public static void main(String [] args){
3
4     for (int i=0; i<10; i=i+1){
5       System.out.println(i);
6     }
7
8   }
9 }
```

Listing 3.22: ForTest.java

Die Reihenfolge, in der die verschiedenen Teile der `for`-Schleife durchlaufen werden, wirkt erst etwas verwirrend, ergibt sich aber natürlich aus der Herleitung der `for`-Schleife aus der vorgeprüften `while`-Schleife:

Als erstes wird genau einmal die Initialisierung der Schleifenvariablen ausgeführt. Anschließend wird die Bedingung geprüft. Abhängig davon wird der Schleifenrumpf ausgeführt. Als letztes wird die Weiterschaltung ausgeführt, bevor wieder die Bedingung geprüft wird.

Die nun schon hinlänglich bekannte Methode `summe` stellt sich

in der Version mit der `for`-Schleife wie folgt dar:

```
1 class Summe3 {
2   public static int summe(int n){
3
4     int erg = 0 ;                    // Feld für Ergebnis
5
6     for (int j = n;j>0;j=j-1){       // j läuft von n bis 1
7       erg = erg + j;                 // akkumuliere das Ergebnis
8     }
9
```

```
10      return erg;
11    }
12  }
```

Listing 3.23: Summe3.java

Beim Vergleich mit der `while`-Version erkennt man, wie sich die Schleifensteuerung im Kopf der `for`-Schleife nun gebündelt an einer syntaktischen Stelle befindet.

Die drei Teile des Kopfes einer `for`-Schleife können auch leer sein. Dann wird in der Regel an einer anderen Stelle der Schleife entsprechender Code zu finden sein. So können wir die Summe auch mit Hilfe der `for`-Schleife so schreiben, dass die Schleifeninitialisierung und Weiterschaltung vor der Schleife bzw. im Rumpf durchgeführt wird:

```
1  class Summe4 {
2    public static int summe(int n){
3
4      int erg = 0 ;              // Feld für Ergebnis.
5      int j   = n;               // Feld zur Schleifenkontrolle
6
7      for (;j>0;){               // j läuft von n bis 1
8        erg = erg + j;           // akkumuliere das Ergebnis.
9        j = j-1;                 // verringere Laufzähler
10     }
11
12     return erg;
13   }
14 }
```

Listing 3.24: Summe4.java

Wie man jetzt sieht, ist die `while`-Schleife nur ein besonderer Fall der `for`-Schleife. Obiges Programm ist ein schlechter Programmierstil. Hier wird ohne Not die Schleifensteuerung mit der eigentlichen Anwendungslogik vermischt.

Schleifen innerhalb von Schleifen

Oft kommt man mit einer Schleife allein nicht aus. Wann immer man einen 2-dimensionalen Raum durchlaufen will, gibt es zwei Richtungen, die x-Richtung und die y-Richtung. Es ist dann durch alle (x,y)-Paare zu iterieren. Hierzu sind zwei ineinander verschachtelte Schleifen notwendig. Die äußere Schleife durchläuft dabei die ein Dimension, die innere Schleife die zweite.

Folgendes Beispiel gibt ein kleines Viereck auf der Kommandozeile aus. Hierfür gibt es eine äußere Schleife, die dafür sorgt, dass jede Zeile ausgegeben wird und eine innere Schleife, die für jede Zeile die einzelnen Zeichen ausgibt.

```
class Cross{

  static String mkCross(int width){
    String result = "";
    for (int y = 0; y < width; y = y +1){
      for (int x = 0; x < width; x = x +1){
        result = result + ( (x==y || (x+y) == (width-1)) ? 'X' : 'O' )
    ;
      }
      result = result + '\n';
    }
    return result;
  }

  public static void main(String[] args){
    System.out.print(mkCross(11));
    System.out.print(mkCross(5));
    System.out.print(mkCross(1));
  }

}
```

Listing 3.25: Cross.java

Aufgabe 1 Schreiben Sie eine Klasse `Karo` mit einer statischen Methode:

`static String karo(int columns,int rows)`

Es soll `String` der folgenden Form erzeugt werden (Beispielaufruf mit `karo(20,15)`):

```
XOXOXOXOXOXOXOXOXOXO
OXOXOXOXOXOXOXOXOXOX
XOXOXOXOXOXOXOXOXOXO
OXOXOXOXOXOXOXOXOXOX
XOXOXOXOXOXOXOXOXOXO
OXOXOXOXOXOXOXOXOXOX
XOXOXOXOXOXOXOXOXOXO
OXOXOXOXOXOXOXOXOXOX
XOXOXOXOXOXOXOXOXOXO
OXOXOXOXOXOXOXOXOXOX
XOXOXOXOXOXOXOXOXOXO
OXOXOXOXOXOXOXOXOXOX
XOXOXOXOXOXOXOXOXOXO
OXOXOXOXOXOXOXOXOXOX
XOXOXOXOXOXOXOXOXOXO
```

Aufgabe 2 Schreiben Sie eine Methode `static int quersumme(int n)`, die die Quersumme einer ganzen Zahl berechnet.

Aufgabe 3 Ergänzen Sie die Klasse `Datum` um eine Methode:

`String monthToString()`.

Die Methode soll einen `String` erzeugen, der zeilenweise den Monat des Datums darstellt, nach der folgenden Form:

```
Freitag der 1.11.2013
Sonnabend der 2.11.2013
Sonntag der 3.11.2013
Montag der 4.11.2013
Dienstag der 5.11.2013
Mittwoch der 6.11.2013
Donnerstag der 7.11.2013
Freitag der 8.11.2013
Sonnabend der 9.11.2013
Sonntag der 10.11.2013
Montag der 11.11.2013
Dienstag der 12.11.2013
Mittwoch der 13.11.2013
Donnerstag der 14.11.2013
Freitag der 15.11.2013
Sonnabend der 16.11.2013
Sonntag der 17.11.2013
Montag der 18.11.2013
Dienstag der 19.11.2013
Mittwoch der 20.11.2013
Donnerstag der 21.11.2013
Freitag der 22.11.2013
Sonnabend der 23.11.2013
Sonntag der 24.11.2013
Montag der 25.11.2013
Dienstag der 26.11.2013
Mittwoch der 27.11.2013
Donnerstag der 28.11.2013
Freitag der 29.11.2013
Sonnabend der 30.11.2013
```

Aufgabe 4 (freiwillige Zusatzaufgabe)

Schreiben Sie in der Klasse `Datum` eine Methode `String monthAsHTML()`, die für den Monat in Form einer HTML Tabelle eine wochenweise Anzeige erzeugt, wie im folgenden Beispiel für November 2013:

```
<table>
 <tr><th>Mo</th><th>Di</th><th>Mi</th><th>Do</th><th>Fr</th><th>Sb</th><th>So</th></tr>
 <tr><td></td><td></td><td></td><td></td><td></td><td>1</td><td>2</td><td>3</td></tr>
 <tr><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td></tr>
 <tr><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td></tr>
 <tr><td>18</td><td>19</td><td>20</td><td><b>21</b></td><td>22</td><td>23</td><td>24</td></tr
 <tr><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td></tr>
</table>
```

3.4 Rekursion

Sobald wir die Signatur einer Funktion oder Prozedur definiert haben, dürfen wir sie benutzen, sprich aufrufen. Damit ergibt sich eine sehr mächtige Möglichkeit der Programmierung. Wir können Funktionen bereits in ihren eigenen Rumpf aufrufen. Solche Funktionen werden rekursivRekursion genannt. *Recurrere* ist das lateinische Wort für zurücklaufen. Eine rekursive Funktion läuft während ihrer Auswertung wieder zu sich selbst zurück.

Damit lassen sich wiederholt Programmteile ausführen. Das folgende Programm wird z.B. nicht müde, uns mit dem Wort `young` zu erfreuen.

```
1 class ForeverYoungRec{
2   static void foreverYoung(){
3     System.out.println("young");
4     foreverYoung();
5   }
6   public static void main(String[] args){
7     foreverYoung();
8   }
9 }
```

Listing 3.26: ForeverYoungRec.java

Die Funktion `main` ruft die Funktion `foreverYoung` auf. Diese druckt einmal das Wort `young` auf die Konsole und ruft sich dann selbst wieder auf. Dadurch wird wieder `young` auf die Konsole geschrieben und so weiter. Wir haben ein endlos laufendes Programm. Tatsächlich endlos? Lassen sie es mal möglichst lange auf Ihrem Rechner laufen.

Was zunächst wie eine Spielerei anmutet, kann verfeinert werden, indem mit Hilfe eines Arguments mitgezählt wird, wie oft die Prozedur bereits rekursiv aufgerufen wurde:

```
1  class HalloZaehler{
2    static void halloZaehler(int i){
3      System.out.println("hallo "+i);
4      halloZaehler(i+1);
5    }
6
7    public static void main(String[] args){
8      halloZaehler(1);
9    }
10 }
```

Listing 3.27: HalloZaehler.java

Auch dieses Programm läuft endlos.

Wie auch bereits bei Schleifen können wir über eine Bedingung dafür sorgen, dass der rekursive Aufruf nur unter bestimmten Umständen ausgeführt werden soll und damit dafür sorgen, dass die Rekursion irgend einmal terminiert.

```
class Rek{
  static void hallo(){
    System.out.println("hallo");
  }

  static void nMalHallo(int n) {
    if (n>0){
      hallo();
      nMalHallo(n-1);
    }
  }

  public static void main(String[] args){
    nMalHallo(5);
  }
}
```

Listing 3.28: Rek.java

Nach soviel esoterisch anmutenden Ausflügen in die Theorie, wollen wir nun auch eine rekursive Funktion schreiben, die etwas interessantes berechnet. Hierzu schreiben wir einmal die Fakultätsfunktion, die mathematisch definiert ist als:

$$fac(n) = \begin{cases} 1 & \text{für } n \leq 0 \\ n * fac(n-1) & \text{für } n > 0 \end{cases}$$

Diese Definition lässt sich direkt in ein Java Programm umsetzen:

```
class Factorial{
  static long fac(long n){
    if (n<=0) return 1;
    return n*fac(n-1);
  }

  public static void main(String[] args){
    System.out.println(fac(5));
    System.out.println(fac(10));
    System.out.println(fac(15));
    System.out.println(fac(20));
    System.out.println(fac(25));
  }
}
```

Listing 3.29: Fac.java

Wir können dieses Programm von Hand ausführen, indem wir den Methodenaufruf für `fac` für einen konkreten Parameter `i` durch die für diesen Wert zutreffende Alternative der Bedingungsabfrage ersetzen. Wir kennzeichnen einen solchen Ersetzungsschritt durch einen Pfeil →:

```
fac(4)

→4*fac(4-1)

→4*fac(3)

→4*(3*fac(3-1))

→4*(3*fac(2))

→4*(3*(2*fac(2-1)))

→4*(3*(2*fac(1)))

→4*(3*(2*(1*fac(1-1))))

→4*(3*(2*(1*fac(0))))

→4*(3*(2*(1*1)))

→4*(3*(2*1))

→4*(3*2)

→4*6

→24
```

3.4.1 Rekursion und Schleifen

Schleifen und Rekursion sind beide dazu geeignet, um Code-Teile wiederholt auszuführen. Man kann auch argumentieren, dass Schleifen ein zusätzliches Konstrukt von Programmiersprachen sind, mit denen sich Rekursion optimierter ausführen lassen können. Betrachten wir die for-Schleife, so lässt sich diese eigentlich immer direkt in eine rekursive Methode umschreiben.

Statt der Schleifenvariablen benötigt die rekursive Methode einen Parameter, der für die Steuerung der Rekursion in einer Abbruchbedingung verwendet wird. Der nächste Durchlauf wird durch den rekursiven Aufruf getätigt. Hierbei wird als neuer Wert der Parameter für die Rekursion übergeben. Eine if-Bedingung sorgt dafür, dass nur unter bestimmten Bedingungen ein weiterer rekursiver Aufruf kommt.

Eine Rekursion, die wie eine for-Schleife über einen Zahlenbereich geht, lässt sich damit wie folgt umsetzen:

```
1  class For{
2    static  void forRek(int from,int to){
3      if (from<=to){
4        System.out.println("hallo "+from );
5        forRek(from+1,to);
6      }
7    }
```

Listing 3.30: ForRek.java

Wir nutzen aus, dass in Java Methoden überladen werden können. Zwei Methoden sind überladen, wenn sie denselben Methodennamen haben, aber unterschiedliche Parameter. Wir schreiben zunächst eine weitere überladene Version von `forRek`, die nur einen Parameter hat, der die obere Grenze der Rekursion angeben soll. Diese überladene Methode ruft direkt die erste Version auf, in dem der erste der beiden Parameter auf 1 gesetzt wird.

```
8    static  void forRek(int to){
9      forRek(1,to);
10   }
```

Listing 3.31: ForRek.java

das lässt sich auch noch ein zweites Mal machen. Dieses Mal hat die überladene Version gar keinen Parameter.

```
11   static void forRek(){
12     forRek(10);
13   }
```

Listing 3.32: ForRek.java

Hier wird auch ein Standardwert genommen. Ruft man die Version ohne Parameter auf, so läuft die Rekursion 10-fach und es wird 10 Mal das Wort `hallo` auf der Kommandozeile ausgegeben.

```
14   public static void main(String[] args){
15     forRek();
16   }
```

```
17 }
```

Listing 3.33: ForRek.java

Codeblöcke als Parameter in Java 8

Die rekursive Methode im obigen Abschnitt, die eine for-Schleife auf rekursive Weise simuliert ist nicht sehr allgemein. Sie ist nur in der Lage für eine bestimmte Anzahl den Befehl `System.out.println("hallo")` auszuführen. Mit der Version 1.8 von Java, die für das Jahr 2014 angekündigt ist und derzeit als Prototyp existiert, wird es möglich sein, Code-Fragmente als Parameter zu übergeben. Für solche Code-Fragmente wird dann ein Typ benutzt, der Typ `Runnable`.

Wir können dann die Methoden um einen weiteren Parameter erweitern. Dieser ist dann vom typ `Runnable`.

```
1  class For8{
2    static  void forRek(Runnable c){
3      forRek(1,c);
4    }
5
6    static  void forRek(int from,Runnable c){
7      forRek(from,10,c);
8    }
9
10   static  void forRek(int from,int to,Runnable c){
```

Listing 3.34: For8.java

Statt nun im eigentlichen Rumpf der rekursiven Methode, den Befehl `System.out.println("hallo")` auszuführen, wird nun für das übergebene Objekt des Typs `Runnable` die Methode `run()` aufgerufen.

```
11     if (from<=to){
12       c.run();
13       forRek(from+1,to,c);
14     }
15   }
```

Listing 3.35: For8.java

Dann wird es in Java 1.8 eine Syntax geben, um Code-Fragmente als Blöcke an Methoden als Parameter zu übergeben. Diese ist in diesem Fall `()->` gefolgt von einem in geschweiften Klammern eingeschlossenen Code-Block. Damit kann jetzt die rekursive Methode genutzt werden, um verschiedene Befehle mehrfach auszuführen.

```
16    public static void main(String[] args){
17      forRek(()->{System.out.println("hallo");});
18
19      forRek(5,9,()->{System.out.println("welt");});
20    }
21  }
```

Listing 3.36: For8.java

Die Änderungen, die mit Java 1.8 in die Sprache eingefügt werden sind gewaltig und erlauben einen vollkommen neuen Programmierstil und eine Methodik, die stärker das funktionale Programmierparadigma verfolgt. Wir werden in diesem Modul aber vorerst noch davon absehen, uns mit die Sprachumfang von Java 1.8 zu beschäftigen und uns erst im zweiten Semester im Rahmen des Moduls »Programmiermethoden und Techniken« damit beschäftigen.

3.4.2 Einsatz von Rekursion

Wie wir gesehen haben, lassen sich mit Rekursion die gleichen Probleme lösen, wie mit der Iteration über Schleifen. Wann sind also besser Schleifen, wann rekursive Lösungen vorzuziehen. Wer die beiden konkurrierenden Versionen von `ForeverYoung` gestartet hat, wird festgestellt haben, dass die rekursive Version relativ schnell abstürzt, während die iterative Version anscheinend endlos läuft.

Tatsächlich haben rekursive Methoden ein Problem: bei jedem Methodenaufruf ist auf einem Speicher abzulegen, von welcher Programmzeile mit was für konkreten Argumenten die Methode aufgerufen wurde. Diese Information wächst bei einer Rekursion schnell an, bis nicht mehr genügend Speicherplatz vorhanden ist, diese zu speichern. Dann stürzt das Programm ab. Daher ist für normale Wiederholungen immer die Schleife vorzuziehen. (Es sei denn man benutzt eine Programmiersprache, deren Compiler Rekursionen optimiert übersetzt.) Wir werden aber in Laufe des Moduls Datentypen kennenlernen, die in sich bereits rekursiv definiert sind und für die rekursive Lösungen adäquat sein können.

Aufgabe 1 Im Bestseller *Sakrileg (der Da Vinci Code)* spielen die Fibonaccizahlen eine Rolle. Für eine natürliche Zahl n ist ihre Fibonaccizahl definiert durch:

$$\text{fib}(n) = \begin{cases} n & \text{für } n \leq 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{für } n > 1 \end{cases}$$

a) Rechnen Sie von Hand den Aufruf fib(5) auf einem Papier aus.

b) Programmieren Sie eine Methode `static int fib(int n)`, die für eine Zahl, die entsprechende Fibonaccizahl zurück gibt.

Aufgabe 2 Schreiben Sie jetzt eine Klasse `Fib`. Objekte dieser Klasse sollen in der Lage sein, nacheinander alle Fibonaccizahlen zu generieren. Hierzu habe die

Klasse zwei Felder vom Typ `int`: `n1` und `n2`. Bei Erzeugung des Objektes sollen diese immer auf `n2==0` und `n1==1` gesetzt sein. Die Klasse soll eine Methode

```
int nextFib()
```

enthalten, die jeweils die nächste Fibonaccizahl generiert. Mehrfacher Aufruf der Methode soll also folgende Zahlenfolge erzeugen:

$1, 1, 2, 3, 5, 8, 13, 21, 34 \ldots$

Ein Beispielaufruf wäre also:

```
1 Fib f = new Fib();
2 System.out.println(f.nextFib());
3 System.out.println(f.nextFib());
4 System.out.println(f.nextFib());
5 System.out.println(f.nextFib());
6 System.out.println(f.nextFib());
```

Aufgabe 3 Ergänzen Sie die Klasse `GeometricObject` aus dem zweiten Übungsblatt um folgende Methoden:

a) `double size()`

Es soll das Produkt aus Höhe und Weite errechnet werden.

b) `boolean isLargerThan(GeometricObject that)`

Soll ausdrücken, dass die Größe von `this` größer ist als vom `that`-Objekt.

c) `boolean isAbove(GeometricObject that)`

das Ergebnis soll wahr sein, wenn das übergebene Objekt eine y-Position hat die größer ist als die Summe aus der y-Position des `this`-Objektes und dessen Höhe.

d) `boolean isUnderneath(GeometricObject that)`

das Ergebnis soll wahr sein, wenn die Summer aus y-Position und Höhe des übergebenen Objekt kleiner ist als die y-Position des `this`-Objektes.

e) `boolean isLeftOf(GeometricObject that)`

das Ergebnis soll wahr sein, wenn das `this`-Objekt auf der x-Achse komplett weiter links als das `that`-Objekt liegt.

f) `boolean isRightOf(GeometricObject that)`

das Ergebnis soll wahr sein, wenn das `this`-Objekt auf der x-Achse komplett weiter rechts als das `that`-Objekt liegt.

g) `boolean touches(GeometricObject that)`

Soll genau dann wahr sein, wenn die beiden Objekte sich in irgendeiner Weise auf dem Koordinatensystem überschneiden.

Aufgabe 4 Ergänzen Sie die Klasse `GeometricObject` um ein weiteres Feld des Typs `Vertex`. Dieses soll die Bewegung des Objektes darstellen.

a) Überladen Sie den Konstruktor der Klasse `GeometricObject`. Wenn es nur einen Parameter des Typs `Vertex` hat, soll dieses die Ecke des Objektes werden. Die Bewegung sei dann in beiden Dimensionen auf 0 gesetzt.

b) Ergänzen Sie `GeometricObject` um eine Methode:

```
void move()
```

Wird sie aufgerufen, soll die Ecke um den Vertex, der die Bewegung ausdrückt, verschoben werden.

Kapitel 4

Weiterführende Konzepte der Objektorientierung

4.1 Vererbung

Eines der grundlegendsten Ziele der objektorientierten Programmierung ist die Möglichkeit, bestehende Programme um neue Funktionalität erweitern zu können. Hierzu bedient man sich der Vererbung. Bei der Definition einer neuen Klassen hat man die Möglichkeit, anzugeben, dass diese Klasse alle Eigenschaften von einer bestehenden Klasse erbt.

Wir haben in einer früheren Übungsaufgabe die Klasse `Person` geschrieben:

```
1  class Person2 {
2
3    String name ;
4    String address;
5
6    Person2(String name, String address){
7      this.name = name;
8      this.address = address;
9    }
10
11   public String toString(){
12     return name+", "+address;
13   }
14 }
```

Listing 4.1: Person2.java

Wenn wir zusätzlich eine Klasse schreiben wollen, die nicht beliebige Personen speichern kann, sondern Studenten, die als zusätzliche Information noch eine Matrikelnummer haben, so stellen wir fest, dass wir wieder Felder für den Namen und die Adresse anlegen müssen; d.h. wir müssen die bereits in der Klasse `Person` zur Verfügung gestellte Funktionalität ein weiteres Mal schreiben:

```
1  class StudentOhneVererbung {
2
3    String name ;
4    String address;
5    int matrikelNummer;
6
7    StudentOhneVererbung(String name, String address,int nr){
8      this.name = name;
9      this.address = address;
10     matrikelNummer = nr;
11   }
12
13   public String toString(){
14     return     name + ", " + address
15             + " Matrikel-Nr.: " + matrikelNummer;
16   }
17 }
```

Listing 4.2: StudentOhneVererbung.java

Mit dem Prinzip der Vererbung wird es ermöglicht, diese Verdoppelung des Codes, der bereits für die Klasse `Person` geschrieben wurde, zu umgehen.

Wir werden in diesem Kapitel schrittweise eine Klasse `Student` entwickeln, die die Eigenschaften erbt, die wir in der Klasse `Person` bereits definiert haben.

Zunächst schreibt man in der Klassendeklaration der Klasse `Student`, dass deren Objekte alle Eigenschaften der Klasse `Person` erben. Hierzu wird das Schlüsselwort `extends` verwendet:

```
1  class Student extends Person2 {
```

Listing 4.3: Student.java

Mit dieser `extends`-Klausel wird angegeben, dass die Klasse von einer anderen Klasse abgeleitet wird und damit deren Eigenschaften erbt. Jetzt brauchen die Eigenschaften, die schon in der Klasse `Person` definiert wurden, nicht mehr neu definiert zu werden.

Mit der Vererbung steht ein Mechanismus zur Verfügung, der zwei primäre Anwendungen hat:

- Erweitern: zu den Eigenschaften der Oberklasse werden weitere Eigenschaften hinzugefügt. Im Beispiel der Studentenklasse soll das Feld `matrikelNummer` hinzugefügt werden.
- Verändern: eine Eigenschaft der Oberklasse wird umdefiniert. Im Beispiel der Studentenklasse soll die Methode `toString` der Oberklasse in ihrer Funktionalität verändert werden.

Es gibt in Java für eine Klasse immer nur genau eine direkte Oberklasse. Eine sogenannte multiple Erbung ist in Java nicht möglich.[1] Es gibt immer maximal eine `extends`-Klausel in einer Klassendefinition.

4.1.1 Hinzufügen neuer Eigenschaften

Unser erstes Ziel der Vererbung war, eine bestehende Klasse um neue

Eigenschaften zu erweitern. Hierzu können wir jetzt einfach mit der `extends`-Klausel angeben, dass wir die Eigenschaften einer Klasse erben. Die Eigenschaften, die wir zusätzlich haben wollen, lassen sich schließlich wie gewohnt deklarieren:

```
2   int matrikelNummer;
```

Listing 4.4: Student.java

Hiermit haben wir eine Klasse geschrieben, die drei Felder hat: `name` und `adresse`, die von der Klasse `Person` geerbt werden und zusätzlich das Feld `matrikelNummer`. Diese drei Felder können für Objekte der

Klasse `Student` in gleicher Weise benutzt werden:

```
3 String writeAllFields(Student s){
4    return s.name+" "+s.address+" "+s.matrikelNummer;
5 }
```

Listing 4.5: Student.java

Ebenso so wie Felder lassen sich Methoden hinzufügen. Z.B. eine Methode, die die Matrikelnummer als Rückgabewert hat:

```
6 int getMatrikelNummer(){
7   return matrikelNummer;
8 }
```

Listing 4.6: Student.java

4.1.2 Überschreiben bestehender Eigenschaften

Unser zweites Ziel ist, durch Vererbung eine Methode in ihrem Verhalten zu verändern. In unserem Beispiel soll die Methode `toString` der Klasse `Person` für Studentenobjekte so geändert werden, dass das Ergebnis auch die Matrikelnummer enthält. Hierzu können wir die entsprechende Methode in der Klasse `Student` einfach neu schreiben:

[1] Dieses ist z.B. in C++ möglich.

```
9   public String toString(){
10    return     name + ", " + address
11             + " Matrikel-Nr.: " + matrikelNummer;
12  }
```

Listing 4.7: Student.java

Obwohl Objekte der Klasse `Student` auch Objekte der Klasse `Person` sind, benutzen sie nicht die Methode `toString` der Klasse `Person`, sondern die neu definierte Version aus der Klasse `Student`.

Um eine Methode zu überschreiben, muss sie dieselbe Signatur bekommen, die sie in der Oberklasse hat.

4.1.3 Konstruktion

Um für eine Klasse konkrete Objekte zu konstruieren, braucht die Klasse entsprechende Konstruktoren. In unserem Beispiel soll jedes Objekt der Klasse `Student` auch ein Objekt der Klasse `Person` sein. Daraus folgt, dass, um ein Objekt der Klasse `Student` zu erzeugen, es auch notwendig ist, ein Objekt der Klasse `Person` zu erzeugen. Wenn wir also einen Konstruktor für `Student` schreiben, sollten wir sicherstellen, dass mit diesem auch ein gültiges Objekt der Klasse `Person` erzeugt wird. Hierzu kann man den Konstruktor der Oberklasse aufrufen. Dieses geschieht mit dem Schlüsselwort `super`. `super` ruft den Konstruktor der Oberklasse auf:

```
13    Student(String name,String adresse,int nr){
14      super(name,adresse);
15      matrikelNummer = nr;
16    }
17  }
```

Listing 4.8: Student.java

In unserem Beispiel bekommt der Konstruktor der

Klasse `Student` alle Daten, die benötigt werden, um ein Personenobjekt und ein Studentenobjekt zu erzeugen. Als erstes wird im Rumpf des Studentenkonstruktors der Konstruktor der Klasse `Person` aufgerufen. Anschließend wird das zusätzliche Feld der Klasse `Student` mit entsprechenden Daten initialisiert.

Ein Objekt der Klasse `Student` kann wie gewohnt konstruiert werden:

```
1  class TestStudent {
2    public static void main(String [] _){
3      Student s
4       = new Student("Martin Müller","Hauptstraße 2",755423);
```

```
5       System.out.println(s);
6     }
7   }
```

Listing 4.9: TestStudent.java

4.1.4 Zuweisungskompatibilität

Objekte einer Klasse sind auch ebenso Objekte ihrer Oberklasse. Daher

können sie benutzt werden wie die Objekte ihrer

Oberklasse, insbesondere bei einer Zuweisung. Da in unserem Beispiel die Objekte der Klasse `Student` auch Objekte der

Klasse `Person` sind, dürfen diese auch Feldern des Typs `Person` zugewiesen werden:

```
1  class TestStudent1{
2    public static void main(String [] args){
3      Person2 p
4       = new Student("Martin Müller","Hauptstraße",7463456);
5    }
6  }
```

Listing 4.10: TestStudent1.java

Alle Studenten sind auch Personen.

Hingegen die andere Richtung ist nicht möglich: nicht alle Personen sind Studenten. Folgendes Programm wird von Java mit einem Fehler zurückgewiesen:

```
1  class StudentError1{
2    public static void main(String [] args){
3      Student s
4        = new Person2("Martin Müller","Hauptstraße");
5    }
6  }
```

Die Kompilierung dieser Klasse führt zu folgender Fehlermeldung:

```
StudentError1.java:3: incompatible types
found   : Person
required: Student
    Student s = new Person2("Martin Müller","Hauptstraße");
                ^
1 error
```

Java weist diese Klasse zurück, weil eine Person nicht ein Student ist.

Gleiches gilt für den Typ von Methodenparametern. Wenn die Methode

einen Parameter vom Typ `Person` verlangt, so kann man ihm auch Objekte eines spezielleren Typs geben, in unserem Fall der Klasse `Student`.

```
1  class TestStudent2 {
2
3    static void printPerson(Person2 p){
4      System.out.println(p.toString());
5    }
6
7    public static void main(String [] args){
8       Student s
9         = new Student("Martin Müller","Hauptstraße",754545);
10      printPerson(s);
11   }
12 }
```

Listing 4.11: TestStudent2.java

Der umgekehrte Fall ist wiederum nicht möglich. Methoden, die als Parameter Objekte der Klasse `Student` verlangen, dürfen nicht mit Objekten einer allgemeineren Klasse aufgerufen werden:

```
1  class StudentError2{
2
3    static void printStudent(Student s){
4      System.out.println(s.toString());
5    }
6
7    public static void main(String [] args){
8       Person2 p = new Person2("Martin Müller","Hauptstraße");
9       printStudent(p);
10   }
11 }
```

Auch hier führt die Kompilierung zu einer entsprechenden Fehlermeldung:

```
StudentError2.java:9: printStudent(Student) in StudentError2
                      cannot be applied to (Person2)
     printStudent(p);
     ^
1 error
```

4.1.5 Späte Bindung (late binding)

Wir haben gesehen, dass wir Methoden überschreiben können. Interessant ist, wann welche Methode ausgeführt wird. In unserem Beispiel gibt es je eine Methode `toString` in der Oberklasse `Person` als auch in der Unterklasse `Student`.

Welche dieser zwei Methoden wird wann ausgeführt? Wir können dieser Frage experimentell nachgehen:

```
1  class TestLateBinding {
2
3    public static void main(String [] args){
4      Student s = new Student("Martin Müller","Hauptstraße",756456);
5      Person2 p1 = new Person2("Harald Schmidt","Marktplatz");
6
7      System.out.println(s.toString());
8      System.out.println(p1.toString());
9
10     Person2 p2 = new Student("Martin Müller","Hauptstraße",756456);
11     System.out.println(p2.toString());
12   }
13 }
```

Listing 4.12: TestLateBinding.java

Dieses Programm erzeugt folgende Ausgabe:

```
sep@swe10:~/fh/> java TestLateBinding
Martin Müller, Hauptstraße Matrikel-Nr.: 756456
Harald Schmidt, Marktplatz
Martin Müller, Hauptstraße Matrikel-Nr.: 756456
```

Die ersten beiden Ausgaben entsprechen sicherlich den Erwartungen: es wird eine Student und anschließend eine Person ausgegeben. Die dritte Ausgabe ist interessant. Obwohl der Befehl:

```
1 System.out.println(p2.toString());
```

die Methode `toString` auf einem Feld vom Typ `Person` ausführt, wird die Methode `toString` aus der Klasse `Student` ausgeführt. Dieser Effekt entsteht, weil das Objekt, das im Feld `p2` gespeichert wurde, als Student und nicht als Person erzeugt wurde. Die Idee der Objektorientierung ist, dass die Objekte die Methoden in sich enthalten. In unserem Fall enthält das Objekt im Feld `p2` seine

eigene `toString`-Methode. Diese wird ausgeführt.

Der Ausdruck `p2.toString()` ist also zu lesen als: Objekt, das in Feld `p2` gespeichert ist, führe bitte deine Methode `toString` aus.

Da dieses Objekt, auch wenn wir es dem Feld nicht ansehen, ein Objekt der Klasse `Student` ist, führt es die entsprechende Methode der Klasse `Student` und nicht der Klasse `Person` aus.

Dieses in Java realisierte Prinzip wird als *late binding* bezeichnet.[2]

Aufgabe 1 In dieser Aufgabe sollen Sie eine Gui-Klasse benutzen und ihr eine eigene Anwendungslogik übergeben.

Gegeben seien die folgenden Javaklassen, wobei Sie die Klasse `Dialogue` nicht zu analysieren oder zu verstehen brauchen:

- ```
1 package name.panitz.oose13.dialogue;
2 public class ButtonLogic {
3   public String getDescription(){
4   return "in Großbuchstaben umwandeln";
5   }
6   public String eval(String x){return x.toUpperCase();}
7 }
```

Listing 4.13: ButtonLogic.java

- ```
1  package name.panitz.oose13.dialogue;
2  import javax.swing.*;
3  import java.awt.event.*;
4  import java.awt.*;
5  public class Dialogue extends JFrame{
6    final JTextField inputField  = new JTextField(20) ;
7    final JTextField outputField = new JTextField(20) ;
8    final JPanel p = new JPanel();
9
10   public Dialogue(final ButtonLogic logic){
11     final JButton button=new JButton(logic.getDescription());
12     button.addActionListener(
13       ev ->outputField.setText(logic.eval(inputField.getText
       ().trim()))
14     );
15     p.setLayout(new BorderLayout());
16     p.add(inputField,BorderLayout.NORTH);
17     p.add(button,BorderLayout.CENTER);
18     p.add(outputField,BorderLayout.SOUTH);
19     getContentPane().add(p);
20     pack();
21     setVisible(true);
22   }
23 }
```

[2] Achtung: *late binding* funktioniert in Java nur bei Methoden, nicht bei Feldern.

Listing 4.14: Dialogue.java

```java
package name.panitz.oose13.dialogue;
class TestDialogue {
  public static void main(String [] _){
    new Dialogue(new ButtonLogic());
  }
}
```

Listing 4.15: TestDialogue.java

a) Übersetzen Sie die drei Klassen und starten Sie das Programm.

b) Schreiben Sie eine Unterklasse der Klasse `ButtonLogic`. Sie sollen dabei die Methoden `getDescription` und `eval` so überschreiben, dass der Eingabestring in Kleinbuchstaben umgewandelt wird. Schreiben Sie eine Hauptmethode, in der Sie ein Objekt der Klasse `Dialogue` mit einem Objekt Ihrer Unterklasse von `ButtonLogic` erzeugen.

c) Schreiben Sie jetzt eine Unterklasse der Klasse `ButtonLogic`, so dass Sie im Zusammenspiel mit der Guiklasse `Dialogue` ein Programm erhalten, das für den im Eingabefeld eingegebenen String die Länge im Ausgabefeld anzeigt.

d) Schreiben Sie jetzt eine Unterklasse der Klasse `ButtonLogic`, so dass Sie im Zusammenspiel mit der Guiklasse `Dialogue` ein Programm erhalten, das für den im Eingabefeld eingegebenen String anzeigt, ob dieser einen Teilstring `"depp"` enthält.

e) Schreiben Sie jetzt eine Unterklasse der Klasse `ButtonLogic`, so dass Sie im Zusammenspiel mit der Guiklasse `Dialogue` ein Programm erhalten, in dem man in dem Eingabefeld eine Jahreszahl eingibt und auf Knopfdruck im Ausgabefeld das Datum des Ostersonntag in diesem Jahr angezeigt wird.

f) Benutzen Sie jetzt die folgende Klasse, um Ihre obigen Unterklassen von `ButtonLogic` mit einem Benutzerschnittstelle auf der Kommandozeile zu verwenden:

```java
package name.panitz.oose13.dialogue;
import java.io.*;

public class ConsoleDialogue {
  final ButtonLogic logic;

  public ConsoleDialogue(ButtonLogic logic) {
    this.logic = logic;
  }
```

```
10
11    public void run(){
12      BufferedReader in
13       = new BufferedReader(new InputStreamReader(System.in));
14      do {
15        try {
16          System.out.println(logic.getDescription()
17                    +"\n(Ende mit 'good bye')");
18          String command = in.readLine();
19          if (command.equals("good bye"))break;
20          System.out.println(logic.eval(command));
21        } catch (IOException e) {
22          System.out.println(e);
23        }
24      }while (true);
25    }
26    public static void main(String[] args) {
27      new ConsoleDialogue(new ButtonLogic()).run();
28    }
29  }
```

Listing 4.16: ConsoleDialogue.java

Die Klasse `Object`

Eine berechtigte Frage ist, welche Klasse die Oberklasse für eine Klasse ist, wenn es keine `extends`-Klausel gibt. Bisher haben wir nie eine entsprechende Oberklasse angegeben.

Java hat in diesem Fall eine Standardklasse: `Object`. Wenn nicht explizit eine Oberklasse angegeben wird, so ist die Klasse `Object` die direkte Oberklasse. Weil die `extends`-Relation transitiv ist, ist schließlich jede Klasse eine Unterklasse der Klasse `Object`. Insgesamt bilden alle Klassen, die in Java existieren, eine Baumstruktur, deren Wurzel die Klasse `Object` ist.

Es bewahrheitet sich die Vermutung über objektorientierte Programmierung, dass alles als Objekt betrachtet wird.[3] Es folgt insbesondere, dass jedes Objekt die Eigenschaften hat, die in der Klasse `Object` definiert wurden. Ein Blick in die Java API Documentation zeigt, dass zu diesen Eigenschaften auch die Methode `toString` gehört, wie wir sie bereits einige mal geschrieben haben. Jetzt erkennen wir, dass wir diese Methode dann überschrieben haben. Auch wenn wir für eine selbstgeschriebene Klasse die Methode `toString` nicht definiert haben, existiert eine solche Methode. Allerdings ist deren Verhalten selten ein für unsere Zwecke geeignetes.

Die Eigenschaften, die alle Objekte haben, weil sie in der Klasse `Object` definiert

[3] Java kennt acht eingebaute primitive Typen für Zahlen, Wahrheitswerte und Buchstaben. Diese sind zwar keine Objekte, werden notfalls von Java aber in entsprechende Objektklassen automatisch konvertiert.

sind, sind äußerst allgemein. Sobald wir von einem Object nur noch wissen, dass es vom Typ `Object` ist, können wir kaum noch spezifische Dinge mit ihm anfangen.

Die Methode `equals` Eine weitere Methode, die in der Klasse `Object` definiert ist, ist die Methode `equals`. Sie hat folgende Signatur:

```
1 public boolean equals(Object other)
```

Wenn man diese Methode überschreibt, so kann definiert werden, wann zwei Objekte einer Klasse als gleich angesehen werden sollen. Für

Personen würden wir gerne definieren, dass zwei Objekte dieser Klasse gleich sind, wenn sie ein und denselben Namen und ein und dieselbe Adresse haben. Mit unseren derzeitigen Mitteln läßt sich dieses leider nicht ausdrücken. Wir würden gerne die `equals`-Methode wie folgt überschreiben:

```
1 public boolean equals(Object other){
2   return         this.name.equals(other.name)
3               && this.adresse.equals(other.adresse);
4 }
```

Dieses ist aber nicht möglich, weil für das Objekt `other`, von dem wir nur wissen, dass es vom Typ `Object` ist, keine Felder `name` und `adresse` existieren.

Um dieses Problem zu umgehen, sind Konstrukte notwendig, die von allgemeineren Typen wieder zu spezielleren Typen führen. Ein solches Konstrukt lernen wir in den folgenden Abschnitten kennen.

Test auf Klasse mit `instanceof`

Wie wir oben gesehen haben, können wir zu wenige Informationen über den Typen eines Objektes haben. Objekte wissen aber selbst, von welcher Klasse sie einmal erzeugt wurden. Java stellt einen binären Operator zur Verfügung, der erlaubt, abzufragen, ob ein Objekt zu einer Klasse gehört. Dieser Operator heißt `instanceof`. Er hat links ein Objekt und rechts einen Klassennamen. Das Ergebnis ist ein bool'scher Wert, der genau dann wahr ist, wenn das Objekt eine Instanz der Klasse ist.

```
1 class InstanceOfTest {
2   public static void main(String [] str){
3      Person2 p1 = new Person2("Strindberg","Skandinavien");
4      Person2 p2 = new Student("Ibsen","Skandinavien",789565);
```

```
5       if (p1 instanceof Student)
6         System.out.println("p1 ist ein  Student.");
7       if (p2 instanceof Student)
8         System.out.println("p2 ist einStudent.");
9       if (p1 instanceof Person2)
10        System.out.println("p1 ist eine Person.");
11      if (p2 instanceof Person2)
12        System.out.println("p2 ist eine Person.");
13    }
14  }
```

Listing 4.17: InstanceOfTest.java

An der Ausgabe dieses Programms kann man erkennen, dass ein `instanceof`-Ausdruck wahr wird, wenn das Objekt ein Objekt der Klasse oder aber einer Unterklasse der Klasse des zweiten Operanden ist.

```
sep@swe10:~/fh> java InstanceOfTest
p2 ist einStudent.
p1 ist eine Person.
p2 ist eine Person.
```

Die Typzusicherung

Im letzten Abschnitt haben wir eine Möglichkeit kennengelernt, zu fragen, ob ein Objekt zu einer bestimmten Klasse gehört. Um ein Objekt dann auch wieder so benutzen zu können, dass es zu dieser Klasse gehört, müssen wir diesem Objekt diesen Typ erst wieder zusichern. Im obigen Beispiel haben wir zwar erfragen können, dass das in Feld `p2` gespeicherte Objekt nicht nur eine Person, sondern ein Student ist; trotzdem können wir noch nicht `p2` nach seiner Matrikelnummer fragen. Hierzu müssen wir erst zusichern, dass das Objekt den Typ `Student` hat.

Eine Typzusicherung in Java wird gemacht, indem dem entsprechenden Objekt in Klammer der Typ vorangestellt wird, den wir ihm zusichern wollen:

```
1  class CastTest {
2    public static void main(String [] str){
3      Person2 p = new Student("Ibsen","Skandinavien",789565);
4
5      if (p instanceof Student){
6        Student s = (Student)p;
7        System.out.println(s.matrikelNummer);
8      }
9    }
10 }
```

Listing 4.18: CastTest.java

Die Zeile `s = (Student)p;` sichert erst dem Objekt im Feld `p` zu, dass es ein Objekt des Typs `Student` ist, so dass es dann als Student benutzt werden kann. Wir haben den Weg zurück vom Allgemeinen ins Spezifischere gefunden. Allerdings ist dieser Weg gefährlich. Eine Typzusicherung kann fehlschlagen:

```
1 class CastError {
2   public static void main(String [] str){
3     Person2 p = new Person2("Strindberg","Skandinavien");
4     Student s = (Student)p;
5     System.out.println(s.matrikelNr);
6   }
7 }
```

Dieses Programm macht eine Typzusicherung des Typs `Student` auf ein Objekt, das nicht von diesem Typ ist. Es kommt in diesem Fall zu einen Laufzeitfehler:

```
sep@swe10:~/fh> java CastError
Exception in thread "main" java.lang.ClassCastException: Person2
        at CastError.main(CastError.java:4)
```

Die Fehlermeldung sagt, dass wir in Zeile 4 des Programms eine Typzusicherung auf ein Objekt des Typs `Person` vornehmen, die fehlschlägt. Will man solche Laufzeitfehler verhindern, so ist man auf der sicheren Seite, wenn eine Typzusicherung nur dann gemacht wird, nachdem man sich mit einem `instanceof`-Ausdruck davon überzeugt hat, dass das Objekt wirklich von dem Typ ist, den man ihm zusichern will.

Mit den jetzt vorgestellten Konstrukten können wir eine Lösung der Methode `equals` für die Klasse `Person2` mit der erwarteten Funktionalität schreiben:

```
1 public boolean equals(Object other){
2   if (other instanceof Person2){
3     Person2 p = (Person2) other;
4     return this.name.equals(p.name)
5            && this.adresse.equals(p.adresse);
6   }
7   return false;
8 }
```

Nur, wenn das zu vergleichende Objekt auch vom Typ `Person` ist und den gleichen Namen und die gleiche Adresse hat, dann sind zwei Personen gleich.

Mehr über die Gleicheit

`public boolean equals(Object obj)` ist die Methode zum Testen auf die Gleichheit von zwei Objekten. Dabei ist die Gleichheit nicht zu verwechseln mit der Identität, die mit dem Operator `==` getestet wird. Der Unterschied zwischen Identität und Gleichheit läßt sich sehr schön an Strings demonstrieren:

```
1  class EqualVsIdentical {
2
3    public static void main(String [] args){
4      String x = "hallo".toUpperCase();
5      String y = "hallo".toUpperCase();
6
7      System.out.println("x: "+x);
8      System.out.println("y: "+y);
9
10     System.out.println("x==x          -> "+(x==x));
11     System.out.println("x==y          -> "+(x==y));
12     System.out.println("x.equals(x) -> "+(x.equals(x)));
13     System.out.println("x.equals(y) -> "+(x.equals(y)));
14   }
15
16 }
```

Listing 4.19: EqualVsIdentical.java

Die Ausgabe dieses Tests ist:

```
sep@swe10:~/fh/beispiele> java EqualVsIdentical
x: HALLO
y: HALLO
x==x          -> true
x==y          ->false
x.equals(x) ->true
x.equals(y) ->true
sep@swe10:~/fh/beispiele>
```

Obwohl die beiden Objekte `x` und `y` die gleichen Texte darstellen, sind es zwei unabhängige Objekte; sie sind nicht identisch, aber gleich.

Auch in der deutschen Sprache gibt es die subtile Unterscheidung zwischen der Identität und einer Gleichheit. Meinen wir eine Gleichheit benützen wir den Ausdruck ›das Gleiche‹, wollen wir die Identität von einem Objekt ausdrücken, so sprechen wir von ›dasselbe‹. Man kann sich das an einer Alltagssituation gut verdeutlichen. Wenn Sie in einem Restaurant dem Kellner sagen, Sie wollen das gleiche Gericht, wie die Dame am Nebentisch, so wird der Kellner in die Küche gehen und dort einen frischen Teller mit eben den Gericht, dass die Dame am Nebentisch bekommen hat, zubereiten lassen. Es gibt also zwei Teller mit zweimal dem gleichen Gericht. Verlangen Sie hingegen dasselbe, wie die Dame am Nebentisch, so muss der Kellner

streng genommen der Dame den Teller wegnehmen und Ihnen hinstellen. Es gibt also nur ein einziges Objekt. das Gleiche heißt in Java »equals«, dasselbe bedeutet in Java `==`. Nur bei primitiven Typen gibt es keinen Unterschied. Dort stellen die Daten nämlich gar keine Objekte dar. Die Klasse `String` hingegen ist kein primitiver Typ und deshalb ist in der Regel zum Vergleichen von zwei String-Objekten die Methode `equals` aufzurufen und nicht der Operator `==` zu verwenden.

Sofern die Methode `equals` für eine Klasse nicht überschrieben wird, wird die entsprechende Methode aus der Klasse `Object` benutzt. Diese überprüft aber keine inhaltliche Gleichheit. Es ist also zu empfehlen, die Methode `equals` für alle eigenen Klassen, die zur Datenhaltung geschrieben wurden, zu überschreiben. Dabei sollte die Methode immer folgender Spezifikation genügen:

- **Reflexivität**: es sollte immer gelten: `x.equals(x)`
- **Symmetrie**: wenn `x.equals(y)` dann auch `y.equals(x)`
- **Transitivität**: wenn `x.equals(y)` und `y.equals(z)` dann gilt auch `x.equals(z)`
- **Konsistenz**: wiederholte Aufrufe von `equals` auf dieselben Objekte liefern dasselbe Ergebnis, sofern die Objekte nicht verändert wurden.
- nichts gleicht `null`: `x.equals(null)` ist immer falsch.

Aufgabe 1 Überschreiben Sie für die Klassen `Vertex`, `GeometricObject` und alle Unterklassen von `GeometricObject` die Methode `equals`. Überlegen Sie, ob Sie damit dann tatsächlich eine Äquivalenzrelation realisiert haben. Wenn nicht, geben Sie ein Gegenbeispiel von Objekten, für die die `equals`-Methoden gegen eines der Gesetze für Äquivalenzrelationen verstößt.

4.2 Pakete

Java bietet die Möglichkeit, Klassen in Paketen zu sammeln. Die Klassen eines Paketes bilden zumeist eine funktional logische Einheit. Pakete sind hierarchisch strukturiert, d.h. Pakete können Unterpakete haben. Damit entsprechen Pakete Ordnern im Dateisystem. Pakete ermöglichen verschiedene Klassen gleichen Namens, die unterschiedlichen Paketen zugeordnet sind.

4.2.1 Paketdeklaration

Zu Beginn einer Klassendefinition kann eine Paketzugehörigkeit für die Klasse definiert werden. Dieses geschieht mit dem Schlüsselwort `package` gefolgt von dem gewünschten Paket. Die Paketdeklaration schließt mit einem Semikolon.

Folgende Klasse definiert sie dem Paket `testPackage` zugehörig:

```
1 package testPackage;
2 class MyClass {
3 }
```

Listing 4.20: MyClass.java

Unterpakete werden von Paketen mit Punkten abgetrennt. Folgende Klasse wird dem Paket `testPackages` zugeordnet, das ein Unterpaket des Pakets `panitz` ist, welches wiederum ein Unterpaket des Pakets `name` ist:

```
1 package name.panitz.testPackages;
2 class TestPaket {
3   public static void main(String [] args){
4     System.out.println("hello from package \'testpackages\'");
5   }
6 }
```

Listing 4.21: TestPaket.java

Paketnamen werden per Konvention in lateinischer Schrift immer mit Kleinbuchstaben als erstem Buchstaben geschrieben.

Wie man sieht, kann man eine weltweite Eindeutigkeit seiner Paketnamen erreichen, wenn man die eigene Webadresse hierzu benutzt.[4] Dabei wird die Webadresse rückwärts verwendet.

Paketname und Klassenname zusammen identifizieren eine Klasse eindeutig. Jeder Programmierer schreibt sicherlich eine Vielzahl von Klassen `Test`, es gibt aber in der Regel nur einen Programmierer, der diese für das Paket `name.panitz.testPackages` schreibt. Paket- und Klassenname zusammen durch einen Punkt getrennt werden der *vollqualifizierte Name* der Klasse genannt, im obigen Beispiel ist entsprechend der vollqualifizierte Name:

`name.panitz.testPackages.Test`

Der Name einer Klasse ohne die Paketnennung heißt unqualifiziert.

4.2.2 Übersetzen von Paketen

Bei größeren Projekten ist es zu empfehlen, die Quelltexte der Javaklassen in Dateien zu speichern, die im Dateisystem in einer Ordnerstruktur, die der Paketstruktur entspricht, liegen. Dieses ist allerdings nicht unbedingt zwingend notwendig. Hingegen zwingend notwendig ist es, die erzeugten Klassendateien in Ordnern entsprechend der Paketstruktur zu speichern.

[4] Leider ist es in Deutschland weit verbreitet, einen Bindestrich in Webadressen zu verwenden. Der Bindestrich ist leider eines der wenigen Zeichen, die Java in Klassen- und Paketnamen nicht zuläßt.

Der Javainterpreter `java` sucht nach Klassen in den Ordnern entsprechend ihrer Paketstruktur. `java` erwartet also, dass die obige Klasse `Test` in einem Ordner `testPackages` steht, der ein Unterordner des Ordners `panitz` ist, der ein Unterordner des Ordners `tfhberlin` ist. usw. `java` sucht diese Ordnerstruktur von einem oder mehreren Startordnern ausgehend. Die Startordner werden in einer Umgebungsvariablen `CLASSPATH` des Betriebssystems und über den Kommandozeilenparameter `-classpath` festgelegt.

Der Javaübersetzer `javac` hat eine Option, mit der gesteuert

wird, dass `javac` für seine `.class`-Dateien die notwendige Ordnerstruktur erzeugt und die Klassen in die ihren Paketen entsprechenden Ordner schreibt. Die Option heißt `-d`. Dem `-d` ist nachgestellt, von welchem Startordner aus die Paketordner erzeugt werden sollen. Memotechnisch steht das `-d` für *destination*.

Wir können die obige Klasse z.B. übersetzen mit folgendem Befehl auf der Kommandozeile:

```
javac -d . Test.java
```

Damit wird ausgehend vom aktuellem Verzeichnis[5] ein Ordner `de` mit Unterordner `tfhberlin` etc. erzeugt.

4.2.3 Starten von Klassen in Paketen

Um Klassen vom Javainterpreter zu starten, reicht es nicht, ihren Namen anzugeben, sondern der vollqualifizierte Name ist anzugeben. Unsere obige kleine Testklasse wird also wie folgt gestartet:

```
sep@swe10:~/> java name.panitz.testPackages.Test
hello from package 'testpackages'
sep@swe10:~/>
```

Jetzt erkennt man auch, warum dem Javainterpreter nicht die Dateiendung `.class` mit angegeben wird. Der Punkt separiert Paket- und Klassennamen.

Aufmerksame Leser werden bemerkt haben, dass der Punkt in Java durchaus konsistent mit einer Bedeutung verwendet wird: hierzu lese man ihn als *'enthält ein'*. Der Ausdruck:

```
name.panitz.testPackages.Test.main(args)
```

liest sich so als: das Paket `de` enthält ein Unterpaket `tfhberlin`, das ein Unterpaket `panitz` enthält, das ein Unterpaket `testpackages` enthält, das eine Klasse `Test` enthält, die eine Methode `main` enthält.

[5] Der Punkt steht in den meisten Betriebssystemen für den aktuellen Ordner, in dem gerade ein Befehl ausgeführt wird.

4.2.4 Das Java Standardpaket

Die mit Java mitgelieferten Klassen sind auch in Paketen gruppiert. Die Standardklassen wie z.B. `String` und `System` und natürlich auch `Object` liegen im Java-Standardpaket `java.lang`. Java hat aber noch eine ganze Reihe weitere Pakete, so z.B. `java.util`, in dem sich Listenklassen befinden, `java.applet`, in dem Klassen zur Programmierung von Applets auf HTML-Seiten liegen, oder `java.io`, welches Klassen für Eingaben und Ausgaben enthält.

4.2.5 Benutzung von Klassen in anderen Paketen

Um Klassen benutzen zu können, die in anderen Paketen liegen, müssen diese eindeutig über ihr Paket identifiziert werden. Dieses kann dadurch geschehen, dass die Klassen immer vollqualifiziert angegeben werden. Im folgenden Beispiel benutzen wir die Standardklasse `ArrayList`[6] aus dem Paket `java.util`.

```
1  package name.panitz.utilTest;
2  class TestArrayList {
3    public static void main(String [] args){
4      java.util.ArrayList<String> xs = new java.util.ArrayList<String>()
       ;
5      xs.add("friends");
6      xs.add("romans");
7      xs.add("countrymen");
8      System.out.println(xs);
9    }
10 }
```

Listing 4.22: TestArrayList.java

Wie man sieht, ist der Klassenname auch beim Aufruf des Konstruktors vollqualifiziert anzugeben.

4.2.6 Importieren von Paketen und Klassen

Importieren von Klassen

Vollqualifizierte Namen können sehr lang werden. Wenn Klassen, die in einem anderen Paket als die eigene Klasse liegen, unqualifiziert benutzt werden sollen, dann kann dieses zuvor angegeben werden. Dieses geschieht zu Beginn einer Klasse in einer Importanweisung. Nur die Klassen aus dem Standardpaket `java.lang` brauchen nicht explizit durch eine Importanweisung bekannt gemacht zu werden.

[6] ArrayList ist eine generische Klasse, ein Konzept, das wir erst in einem späteren Kapitel kennenlernen werden.

Unsere Testklasse aus dem letzten Abschnitt kann mit Hilfe einer Importanweisung so geschrieben werden, dass die Klasse `ArrayList` unqualifiziert[7] benutzt werden kann:

```
1  package name.panitz.utilTest;
2
3  import java.util.ArrayList;
4
5  class TestImport {
6    public static void main(String [] args){
7      ArrayList<String> xs = new ArrayList<String>();
8      xs.add("friends");
9      xs.add("romans");
10     xs.add("countrymen");
11     System.out.println(xs);
12   }
13 }
```

Listing 4.23: TestImport.java

Es können mehrere Importanweisungen in einer Klasse stehen. So können wir z.B. zusätzlich die Klasse `Vector` importieren:

```
1  package name.panitz.utilTest;
2
3  import java.util.ArrayList;
4  import java.util.Vector;
5
6  class TestImport2 {
7    public static void main(String [] args){
8      ArrayList<String> xs = new ArrayList<String>();
9      xs.add("friends");
10     xs.add("romans");
11     xs.add("countrymen");
12     System.out.println(xs);
13
14     Vector<String> ys = new Vector<String>();
15     ys.add("friends");
16     ys.add("romans");
17     ys.add("countrymen");
18     System.out.println(ys);
19   }
20 }
```

Listing 4.24: TestImport2.java

[7] Aus historischen Gründen wird in diesem Kapitel als Beispiel bereits mit den generischen Klassen ArrayList und Vector ein Konzept benutzt, das erst im nächsten Kapitel erklärt wird.

Importieren von Paketen

Wenn in einem Programm viele Klassen eines Paketes benutzt werden, so können mit einer Importanweisung auch alle Klassen dieses Paketes importiert werden. Hierzu gibt man in der Importanweisung einfach statt des Klassennamens ein * an.

```
1  package name.panitz.utilTest;
2
3  import java.util.*;
4
5  class TestImport3 {
6    public static void main(String [] args){
7      List<String> xs = new ArrayList<String>();
8      xs.add("friends");
9      System.out.println(xs);
10
11     Vector<String> ys = new Vector<String>();
12     ys.add("romans");
13     System.out.println(ys);
14   }
15 }
```

Listing 4.25: TestImport3.java

Ebenso wie mehrere Klassen können auch mehrere komplette Pakete importiert werden. Es können auch gemischt einzelne Klassen und ganze Pakete importiert werden.

4.2.7 Statische Imports

Statische Eigenschaften einer Klasse werden in Java dadurch angesprochen, dass dem Namen der Klasse mit Punkt getrennt die gewünschte Eigenschaft folgt. Werden in einer Klasse sehr oft statische Eigenschaften einer anderen Klasse benutzt, so ist der Code mit deren Klassennamen durchsetzt. Die Javaentwickler haben mit Java 1.5 ein Einsehen. Man kann jetzt für eine Klasse alle ihre statischen Eigenschaften importieren, so dass diese unqualifiziert benutzt werden kann. Die `import`-Anweisung sieht aus wie ein gewohntes Paktimport, nur dass das Schlüsselwort `static` eingefügt ist und erst dem klassennamen der Stern folgt, der in diesen Fall für alle statischen Eigenschaften steht.

Wir schreiben eine Hilfsklasse zum Arbeiten mit Strings, in der wir eine Methode zum umdrehen eines Strings vorsehen:

```
1  package name.panitz.staticImport;
2  public class StringUtil {
3    static public String reverse(String arg) {
4      StringBuffer result = new StringBuffer();
```

```
5       for (char c:arg.toCharArray()) result.insert(0,c);
6       return result.toString();
7     }
8   }
```

Listing 4.26: StringUtil.java

Die Methode `reverse` wollen wir in einer anderen Klasse benutzen. Importieren wir die statischen Eigenschaften von `StringUtil`, so können wir auf die Qualifizierung des Namens der Methode `reverse` verzichten:

```
1  package name.panitz.staticImport;
2  import static name.panitz.staticImport.StringUtil.*;
3  public class UseStringUtil {
4    static public void main(String [] args) {
5      for (String arg:args)
6       System.out.println(reverse(arg));
7    }
8  }
```

Listing 4.27: UseStringUtil.java

Die Ausgabe dieses programms:

```
sep@linux:fh> java -classpath classes/ name.panitz.staticImport.UseStringUtil hallo welt
ollah
tlew
sep@linux:~/fh/java1.5/examples>
```

4.2.8 Sichtbarkeitsattribute

ichtbarkeiten[8] erlauben es, zu kontrollieren, wer auf Klassen und ihre Eigenschaften zugreifen kann. Das *wer* bezieht sich hierbei auf andere Klassen und Pakete.

Sichtbarkeitsattribute für Klassen

Für Klassen gibt es zwei Möglichkeiten der Sichtbarkeit. Entweder darf von überall aus eine Klasse benutzt werden oder nur von Klassen im gleichen Paket. Syntaktisch wird dieses dadurch ausgedrückt, dass der Klassendefinition entweder das Schlüsselwort `public` vorangestellt ist oder aber kein solches Attribut voransteht:

```
1  package name.panitz.p1;
2  public class MyPublicClass {
3  }
```

Listing 4.28: MyPublicClass.java

[8] Man findet in der Literatur auch den Ausdruck *Erreichbarkeiten*.

```
1 package name.panitz.p1;
2 class MyNonPublicClass {
3 }
```

Listing 4.29: MyNonPublicClass.java

In einem anderen Paket dürfen wir nur die als öffentlich deklarierte Klasse benutzen. Folgende Klasse übersetzt fehlerfrei:

```
1 package name.panitz.p2;
2
3 import name.panitz.p1.*;
4
5 class UsePublic {
6   public static void main(String [] args){
7     System.out.println(new MyPublicClass());
8   }
9 }
```

Listing 4.30: UsePublic.java

Der Versuch, eine nicht öffentliche Klasse aus einem anderen Paket heraus zu benutzen, gibt hingegen einen Übersetzungsfehler:

```
1 package name.panitz.p2;
2
3 import name.panitz.p1.*;
4
5 class UseNonPublic {
6   public static void main(String [] args){
7     System.out.println(new MyNonPublicClass());
8   }
9 }
```

Java gibt bei der Übersetzung eine entsprechende gut verständliche Fehlermeldung:

```
sep@swe10:~> javac -d . UseNonPublic.java
UseNonPublic.java:7: name.panitz.p1.MyNonPublicClass is not
public in name.panitz.pantitz.p1;
cannot be accessed from outside package
        System.out.println(new MyNonPublicClass());
                               ^
UseNonPublic.java:7: MyNonPublicClass() is not
public in name.panitz.p1.MyNonPublicClass;
cannot be accessed from outside package
        System.out.println(new MyNonPublicClass());
                           ^
2 errors
sep@swe10:~>
```

Damit stellt Java eine Technik zur Verfügung, die es erlaubt, bestimmte Klassen eines Softwarepaketes als rein interne Klassen zu schreiben, die von außerhalb des Pakets nicht benutzt werden können.

Sichtbarkeitsattribute für Eigenschaften

Java stellt in Punkto Sichtbarkeiten eine noch feinere Granularität zur Verfügung. Es können nicht nur ganze Klassen als nicht-öffentlich deklariert, sondern für einzelne Eigenschaften von Klassen unterschiedliche Sichtbarkeiten deklariert werden.

Für Eigenschaften gibt es vier verschiedene Sichtbarkeiten:

`public`, `protected`, kein Attribut, `private`

Sichbarkeiten hängen zum einem von den Paketen ab, in denen sich die Klassen befinden, darüberhinaus unterscheiden sich Sichtbarkeiten auch darin, ob Klassen Unterklassen voneinander sind. Folgende Tabelle gibt eine Übersicht über die vier verschiedenen Sichtbarkeiten:

Damit kann in einer Klasse auf Eigenschaften mit jeder dieser vier Sichtbarkeiten zugegriffen werden. Wir können die Fälle einmal systematisch durchprobieren. In einer öffentlichen Klasse eines Pakets `p1` definieren wir hierzu vier Felder mit den vier unterschiedlichen Sichtbarkeiten:

```
1  package name.panitz.p1;
2
3  public class VisibilityOfFeatures{
4    private   String s1 = "private";
5              String s2 = "package";
6    protected String s3 = "protected";
7    public    String s4 = "private";
8
9    public static void main(String [] args){
10     VisibilityOfFeatures v = new VisibilityOfFeatures();
11     System.out.println(v.s1);
12     System.out.println(v.s2);
13     System.out.println(v.s3);
14     System.out.println(v.s4);
15   }
16 }
```

Listing 4.31: VisibilityOfFeatures.java

In der Klasse selbst können wir auf alle vier Felder zugreifen.

In einer anderen Klasse, die im gleichen Paket ist, können private Eigenschaften nicht mehr benutzt werden:

```
1  package name.panitz.p1;
2
3  public class PrivateTest
4  {
5
6    public static void main(String [] args){
7      VisibilityOfFeatures v = new VisibilityOfFeatures();
8      //s1 is private and cannot be accessed;
9      //we are in a  different class.
10     //System.out.println(v.s1);
11     System.out.println(v.s2);
12     System.out.println(v.s3);
13     System.out.println(v.s4);
14   }
15 }
```

Listing 4.32: PrivateTest.java

Von einer Unterklasse können unabhängig von ihrem Paket die *geschützten* Eigenschaften benutzt werden. Ist die Unterklasse in einem anderen Paket, können Eigenschaften mit der Sichtbarkeit `package` nicht mehr benutzt werden:

```
1  package name.panitz.p2;
2  import name.panitz.p1.VisibilityOfFeatures;
3
4  public class  PackageTest extends VisibilityOfFeatures{
5
6    public static void main(String [] args){
7      PackageTest v = new PackageTest();
8      //s1 is private and cannot be accessed
9      // System.out.println(v.s1);
10
11     //s2 is package visible and cannot be accessed;
12     //we are in a different package.
13     //System.out.println(v.s2);
14
15     System.out.println(v.s3);
16     System.out.println(v.s4);
17   }
18 }
```

Listing 4.33: PackageTest.java

Von einer Klasse, die weder im gleichen Paket noch eine Unterklasse ist, können nur noch öffentliche Eigenschaften benutzt werden:

```java
package name.panitz.p2;
import name.panitz.p1.VisibilityOfFeatures;

public class ProtectedTest  {

  public static void main(String [] args){
    VisibilityOfFeatures v = new VisibilityOfFeatures();
    //s1 is private and cannot be accessed
    // System.out.println(v.s1);

    //s2 is package visible and cannot be accessed. We are
    //in a different package
    //System.out.println(v.s2);

    //s2 is protected and cannot be accessed.
    //We are not a subclass
    //System.out.println(v.s3);

    System.out.println(v.s4);
  }
}
```

Listing 4.34: ProtectedTest.java

Java wird in seinem Sichtbarkeitskonzept oft kritisiert, und das von zwei Seiten. Einerseits ist es mit den vier Sichtbarkeiten schon relativ unübersichtlich; die verschiedenen Konzepte der Vererbung und der Pakete spielen bei Sichtbarkeiten eine Rolle. Andererseits ist es nicht vollständig genug und kann verschiedene denkbare Sichtbarkeiten nicht ausdrücken.

In der Praxis fällt die Entscheidung zwischen privaten und öffentlichen Eigenschaften leicht. Geschützte Eigenschaften sind hingegen selten. Das Gros der Eigenschaften hat die Standardsichtbarkeit der Paketsichtbarkeit.

Aufgabe 2 Nehmen Sie die Klassen `Vertex` und `GeometricObject` und stecken Sie diese in das Paket `de.hsrm.cs.oose13`.

Aufgabe 3 Schreiben Sie eine Unterklasse `Ellipse` der Klasse `GeometricObject`. Weite und Höhe einer Ellipse sind die beiden Durchmesser der Ellipse. Überladen Sie mehrere Konstruktoren. Überschreiben Sie die Methoden `size` und `toString`.

Aufgabe 4 Schreiben Sie eine Unterklasse `Circle` der Klasse `Ellipse`. Dieses ist eine Ellipse in der beide Durchmesser, also die Breite und die Höhe gleich groß sind.

4.3 Schnittstellen (Interfaces) und abstrakte Klassen

Wir haben schon Situationen kennengelernt, in denen wir eine Klasse geschrieben haben, von der nie ein Objekt konstruiert werden sollte, sondern für die wir nur Unterklassen definiert und instanziiert haben. Die Methoden in diesen Klassen hatten eine möglichst einfache Implementierung; sie sollten ja nie benutzt werden, sondern die überschreibenden Methoden in den Unterklassen. Ein Beispiel für eine solche Klassen war die Klasse `ButtonLogic`, mit der die Funktionalität eines GUIs definiert wurde.

Java bietet ein weiteres Konzept an, mit dem Methoden ohne eigentliche Implementierung deklariert werden können, die Schnittstellen.

4.3.1 Schnittstellen

Schnittstellendeklaration

Eine Schnittstelle sieht einer Klasse sehr ähnlich. Die syntaktischen Unterschiede sind:

- statt des Schlüsselworts `class` steht das Schlüsselwort `interface`.
- die Methoden haben keine Rümpfe, sondern nur eine Signatur.

So läßt sich für unsere Klasse `ButtonLogic` eine entsprechende Schnittstelle schreiben:

```
1 package name.panitz.dialoguegui;
2
3 public interface DialogueLogic {
4   public String getDescription();
5   public String eval(String input);
6 }
```

Listing 4.35: DialogueLogic.java

Schnittstellen sind ebenso wie Klassen mit dem Javaübersetzer zu übersetzen. Für Schnittstellen werden auch Klassendateien mit der Endung `.class` erzeugt.

Im Gegensatz zu Klassen haben Schnittstellen keinen Konstruktor. Das bedeutet insbesondere, dass mit einer Schnittstelle kein Objekt erzeugt werden kann. Was hätte ein solches Objekt auch für ein Verhalten? Die Methoden haben ja gar keinen Code, den sie ausführen könnten. Eine Schnittstelle ist vielmehr ein Versprechen, dass Objekte Methoden mit den in der Schnittstelle definierten Signaturen enthalten. Objekte können aber immer nur über Klassen erzeugt werden.

Implementierung von Schnittstellen

Objekte, die die Funktionalität einer Schnittstelle enthalten, können nur mit Klassen erzeugt werden, die diese Schnittstelle implementieren. Hierzu gibt es zusätzlich zur `extends`-Klausel in Klassen auch noch die Möglichkeit, eine `implements`-Klausel anzugeben.

Eine mögliche Implementierung der obigen Schnittstelle ist:

```
1  package name.panitz.dialoguegui;
2
3  public class ToUpperCase implements DialogueLogic{
4    protected String result;
5
6    public String getDescription(){
7      return "convert into upper cases";
8    }
9    public String eval(String input){
10     result = input.toUpperCase();
11     return result;
12   }
13 }
```

Listing 4.36: ToUpperCase.java

Die Klausel `implements DialogueLogic` verspricht, dass in dieser Klasse für alle Methoden aus der Schnittstelle eine Implementierung existiert. In unserem Beispiel waren zwei Methoden zu implementieren, die Methode `eval` und `getDescription()`.

Im Gegensatz zur `extends`-Klausel von Klassen können in einer `implements`-Klausel auch mehrere Schnittstellen angegeben werden, die implementiert werden.

Definieren wir zum Beispiel ein zweite Schnittstelle:

```
1  package name.panitz.html;
2
3  public interface ToHTMLString {
4    public String toHTMLString();
5  }
```

Listing 4.37: ToHTMLString.java

Diese Schnittstelle verlangt, dass implementierende Klassen eine Methode haben, die für das Objekt eine Darstellung als HTML erzeugen können.

Jetzt können wir eine Klasse schreiben, die die beiden Schnittstellen implementiert.

```
1  package name.panitz.dialoguegui;
2
3  import name.panitz.html.*;
```

```
4
5  public class ToUpper extends     ToUpperCase
6                       implements ToHTMLString, DialogueLogic {
7
8    public String toHTMLString(){
9      return    "<html><head><title>"+getDescription()
10             + "</title></head>"
11             + "<body><b>Small Gui application</b>"
12             + " for convertion of "
13             + " a <b>String</b> into <em>upper</em>"
14             + " case letters.<br></br>"
15             + "The result of your query was: <p>"
16             + "<span style=\"font-family: monospace;\">"
17             + result
18             + "</span></p></body></html>";
19   }
20 }
```

Listing 4.38: ToUpper.java

Schnittstellen können auch einander erweitern. Dieses geschieht dadurch, dass Schnittstellen auch eine `extends`-Klausel haben.

Wir können also auch eine Schnittstelle definieren, die die beiden obigen Schnittstellen zusammenfaßt:

```
1 package name.panitz.dialoguegui;
2
3 import name.panitz.html.*;
4
5 public interface DialogueLogics extends ToHTMLString, DialogueLogic {}
```

Listing 4.39: DialogueLogics.java

Ebenso können wir jetzt eine Klasse ableiten, die diese Schnittstelle implementiert:

```
1 package name.panitz.dialoguegui;
2
3 class UpperConversion extends ToUpper
4                       implements DialogueLogics{}
```

Listing 4.40: UpperConversion.java

Benutzung von Schnittstellen

Schnittstellen sind genauso Typen wie Klassen. Wir kennen jetzt also drei Arten von Typen:

- primitive Typen
- Klassen

- Schnittstellen

Parameter können vom Typ einer Schnittstellen sein, ebenso wie Felder oder Rückgabetypen von Methoden. Die Zuweisungkompatibilität nutzt nicht nur die Unterklassenbeziehung, sondern auch die Implementierungsbeziehung. Ein Objekt der Klasse `C` darf einem Feld des Typs der Schnittstelle `I` zugewiesen werden, wenn `C` die Schnittstelle `I` implementiert.

Im Folgenden eine kleine Gui-Anwendung, die wir im einzelnen noch nicht verstehen müssen. Man beachte, dass der Typ `DialogueLogics` an mehreren Stellen benutzt wird wie ein ganz normaler Klassentyp. Nur einen Konstruktoraufruf mit `new` können wir für diesen Typ nicht machen.

```
1  package name.panitz.dialoguegui;
2  import java.awt.event.*;
3  import java.awt.*;
4  import javax.swing.*;
5  import javax.swing.plaf.basic.*;
6  import javax.swing.text.*;
7  import javax.swing.text.html.*;
8
9  public class HtmlDialogue extends JFrame{
10   final DialogueLogics logic;
11   final JButton button;
12   final JTextField inputField = new JTextField(20) ;
13   final JTextPane outputField = new JTextPane();
14   final JPanel p = new JPanel();
15
16   public HtmlDialogue(DialogueLogics l){
17     outputField.setEditorKit(new HTMLEditorKit());
18     logic = l;
19     button=new JButton(logic.getDescription());
20     button.addActionListener
21      (new ActionListener(){
22         public void actionPerformed(ActionEvent _){
23           logic.eval(inputField.getText().trim());
24           outputField.setText(logic.toHTMLString());
25           pack();
26         }
27       });
28
29     p.setLayout(new BorderLayout());
30     p.add(inputField,BorderLayout.NORTH);
31     p.add(button,BorderLayout.CENTER);
32     p.add(outputField,BorderLayout.SOUTH);
33     getContentPane().add(p);
34     pack();
35     setVisible(true);
36   }
37 }
```

Listing 4.41: HtmlDialogue.java

Schließlich können wir ein Objekt der Klasse `UpperConversion`, die die Schnittstelle `DialogueLogics` implementiert, konstruieren und der Gui-Anwendung übergeben:

```
1 package name.panitz.dialoguegui;
2 public class HtmlDialogueTest {
3   public static void main(String [] args){
4     new HtmlDialogue(new UpperConversion());
5   }
6 }
```

Listing 4.42: HtmlDialogueTest.java

Die Anwendung in voller Aktion kann in Abbildung **??** bewundert werden.

Ein Gui-Dialog mit Html-Ausgabe.

Weitere Eigenschaften von Schnittstellen

Es gibt einige semantische Einschränkungen, die über die syntaktischen Einschränkungen hinausgehen:

- Schnittstellen können nur Schnittstellen, nicht aber Klassen erweitern.
- Jede Methode einer Schnittstelle muss öffentlich sein, braucht also das Attribut `public`. Wenn dieses für eine Methode nicht deklariert ist, so wird Java dieses von selbst hinzufügen. Trotzdem müssen implementierende Klassen diese Methode dann als öffentlich deklarieren. Daher ist es besser, das Attribut `public` auch hinzuschreiben.
- Es gibt keine statischen Methoden in Schnittstellen.
- Jede Methode ist abstrakt, d.h. hat keinen Rumpf. Man kann dieses noch zusätzlich deutlich machen, indem man das Attribut `abstract` für die Methode mit angibt. Allerdings werden mit Java 1.8 sogenannte default-Methoden in Schnittstellen möglich. Diese sind dann nicht mehr abstrakt und haben einen Methodenrumpf.
- Felder einer Schnittstelle sind immer statisch, brauchen also das Attribut `static` und zusätzlich noch das Attribut `final`.

Aufgabe 2 Gegeben Seien folgende Schnittstellen:

```
1 package de.hsrm.cs.oose13;
2 public interface Moveable{
3   void move();
4 }
```

Listing 4.43: Moveable.java

```java
package de.hsrm.cs.oose13;
import java.awt.Graphics;

public interface Paintable{
  void paintMeTo(Graphics g);
}
```

Listing 4.44: Paintable.java

```java
package de.hsrm.cs.oose13;
public interface MoveAndPaintable extends Moveable, Paintable{
}
```

Listing 4.45: MoveAndPaintable.java

Lassen Sie `GeometricObject` und allen Unterklassen auf adäquate Weise die Schnittstelle `MoveAndPaintable` implementieren.

Aufgabe 3 Testen Sie Beispielobjekte all Ihrer Klassen mit folgender Klasse:

```java
package de.hsrm.cs.oose13;
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class ShowMoveable extends JPanel{
  MoveAndPaintable mvpt;
  Timer t;
  public ShowMoveable(MoveAndPaintable mvpt){
    this.mvpt = mvpt;
    t = new Timer(100,new ActionListener(){
      public void actionPerformed(ActionEvent ev){
        move();
        repaint();
      }
    });
  }
  @Override
  public Dimension getPreferredSize(){
    return new Dimension(800,600);
  }
  @Override
  public void paintComponent(Graphics g){
    super.paintComponent(g);
    mvpt.paintMeTo(g);
  }
  public void move(){
    mvpt.move();
  }
  public void show(){
    JFrame f = new JFrame();
```

```
32      f.add(this);
33      f.pack();
34      f.setVisible(true);
35      t.start();
36    }
37  }
```

Listing 4.46: ShowMoveable.java

Hierzu müssen Sie den Konstruktor aufrufen `new ShowMoveable(.....)` und für das so erzeugte Objekt die Methode `show` aufrufen.

Aufgabe 4 Schreiben Sie eine weitere Unterklasse von `GeometricObject`. Diese soll einen Stern darstellen. Ein Stern lässt sich durch folgende Eigenschaften beschreiben:

- einen inneren Radius, für die Einbuchtungen des Strahlen.
- einen äußeren Radius für die Spitzen der Strahlen.
- die Anzahl der Strahlen des Sterns

Benutzen Sie beide der Implementierung der Methode `paintMeTo` die Methode `fillPolygon` des Graphics-Objekts und die Klasse `Polygon` des Standard API.

Kapitel 5

Graphische Benutzeroberflächen mit Swing

Die meisten Programme auf heutigen Rechnersystemen haben eine graphische Benutzeroberfläche (GUI)[1].

Java stellt Klassen zur Verfügung, mit denen graphische Objekte erzeugt und in ihrem Verhalten instrumentalisiert werden können. Es gibt mehrere Bibliotheken in Java, die Klassen bereit stellen, um GUIs zu implementieren. Ganz ursprünglich gab es in Java die AWT Bibliothek. Deren Klassen liegen im Paket `java.awt`. Doch schon nach wenigen Jahren wurde eine neue Bibliothek entwickelt, die sogenannte Swing GUI-Bibliothek. Deren Klassen befinden sich im Paket `javax.swing`. Mittlerweile soll langfristig Swing nicht mehr weiter entwickelt werden, sondern in Zukunft die Bibliothek Java FX der Standard zur GUI Programmierung in Java sein. Es gibt aber auch externe GUI-Bibliotheken. Hier ist insbesondere http://www.eclipse.org/swt/ (`http://www.eclipse.org/swt/`) zu nennen. Auch das google web toolkit, das zur Entwicklung von Webapplikationen von der Firma Google zur Verfügung gestellt wird, enthält eine GUI-Bibliothek. Dort wird die entwickelte Benutzerschnittstelle so übersetzt, dass sie in einem Browser läuft.

Wir werden uns in diesem Kapitel als Beispiel der Swing Bibliothek widmen.

Leider kommt man bei der Swing Programmierung nicht darum herum, auch Klassen aus der AWT Bibliothek zu verwenden. Die beiden Pakete überschneiden sich in der Funktionalität.

- `java.awt`: Dieses ist das ältere Paket zur GUI-Programmierung. Es enthält Klassen für viele graphische Objekte (z.B. eine Klasse `Button`) und Unterpakete, zur Programmierung der Funktionalität der einzelnen Komponenten.
- `javax.swing`: Dieses neuere Paket ist noch universeller und platformunabhängiger als das `java.awt`-Paket. Auch hier finden sich Klassen für unterschiedliche GUI-Komponenten. Sie entsprechen den Klassen aus dem Paket java.awt. Die Klassen haben oft den gleichen Klassennamen wie in `java.awt` jedoch mit einem `J` vorangestellt. So gib es z.B. eine Klasse `JButton`.

[1] GUI ist die Abkürzung für *graphical user interface*. Entsprechend wäre GRABO eine Abkürzung für das deutsche *graphische Benutzeroberfläche*.

Man ist angehalten, sofern man sich für eine Implementierung seines GUIs mit den Paket `javax.swing` entschieden hat, nur die graphischen Komponenten aus diesem Paket zu benutzen; die Klassen leiten aber von Klassen des Pakets `java.awt` ab. Hinzu kommt, dass die Ereignisklassen, die die Funktionalität graphischer Objekte bestimmen, nur in `java.awt` existieren und nicht noch einmal für `javax.swing` extra umgesetzt wurden.

Sind Sie verwirrt? Sie werden es hoffentlich nicht mehr sein, nachdem Sie die Beispiele dieses Kapitels durchgespielt haben.

Zur Programmierung eines GUIs müssen drei fundamentale Fragestellungen gelöst werden:

- Welche graphischen Komponenten stehen zur Verfügung? Knöpfe, Textfelder, Baumdarstellugen, Tabellen....
- Wie können diese Komponenten angeordnet und gruppiert werden? Was gibt es für Layout-Möglichkeiten?
- Wie reagiert man auf Ereignisse, zum Beispiel auf einen Mausklick?

Javas Swing Bibliothek kennt für die Aufgaben:

- Komponenten wie `JButton`, `JTextField`, `JLabel`,...
- Layout Manager und Komponenten, um andere Komponenten zu gruppieren, z.B. `JPanel`
- Event Handler, in denen implementiert wird, wie auf ein Ereignis reagiert werden soll.

In den folgenden Abschnitten, werden wir an ausgewählten Beispielen Klassen für diese drei wichtigen Schritte der GUI-Programmierung kennenlernen.

5.1 Swings GUI-Komponenten

Javas `swing`-Paket kennt drei Arten von Komponenten.

- Top-Level Komponenten
- Zwischenkomponenten
- Atomare Komponenten

Leider spiegelt sich diese Unterscheidung nicht in der Ableitungshierarchie wider. Alle Komponenten leiten schließlich von der Klasse `java.awt.Component` ab. Es gibt keine Schnittmengen, die Beschreiben, dass bestimmte Komponenten atomar oder top-level sind.

Komponenten können Unterkomponenten enthalten; ein Fenster kann z.B. verschiedene Knöpfe und Textflächen als Unterkomponenten enthalten.

5.1.1 Top-Level Komponenten

Eine top-level Komponenten ist ein GUI-Objekt, das weitere graphische Objekte enthalten kann, selbst aber kein graphisches Objekt hat, in dem es enthalten ist. Somit sind top-level Komponenten in der Regel Fenster, die weitere Komponenten als Fensterinhalt haben. Swing top-level Komponenten sind Fenster und Dialogfenster. Hierfür stellt Swing entsprechende Klassen zur Verfügung: `JFrame, JDialog` Eine weitere top-level Komponente steht für *Applets* zur Verfügung: `JApplet`.

Graphische Komponenten haben Konstruktoren, mit denen sie erzeugt werden. Für die Klasse `JFrame` existiert ein parameterloser Konstruktor.

Das minimalste GUI-Programm ist wahrscheinlich folgendes Programm, das ein Fensterobjekt erzeugt und dieses sichtbar macht.

```
1 package name.panitz.oose.swing.example;
2 import javax.swing.*;
3
4 class JF {
5   public static void main(String [] _){
6     new JFrame().setVisible(true);
7   }
8 }
```

Listing 5.1: JF.java

Dieses Programm erzeugt ein leeres Fenster und gibt das auf dem Bildschirm aus.

Die Klasse `JFrame` hat einen zweiten Konstruktor, der noch ein Stringargument hat. Der übergebene String wird als Fenstertiteltext benutzt.

Ein Programm, kann auch mehrere Fensterobjekte erzeugen und sichtbar machen.

```
1 package name.panitz.oose.swing.example;
2 import javax.swing.*;
3
4 class JF2 {
5   public static void main(String [] _){
6     new JFrame("erster Rahmen").setVisible(true);
7     new JFrame("zweiter Rahmen").setVisible(true);
8   }
9 }
```

Listing 5.2: JF2.java

5.1.2 Zwischenkomponenten

Graphische Zwischenkomponenten haben primär die Aufgabe andere Komponenten als Unterkomponenten zu haben und diese in einer bestimmten Weise anzuordnen.

Zwischenkomponenten haben oft keine eigene visuelle Ausprägung. Sie sind dann unsichtbare Komponenten, die als Behälter weiterer Komponenten dienen.

Die gebräuchlichste Zwischenkomponenten ist von der Klasse `JPanel`. Weitere Zwischenkomponenten sind `JScrollPane` und `JTabbedPane`. Diese haben auch eine eigene visuelle Ausprägung.

5.1.3 Atomare Komponenten

Die atomaren Komponenten sind schließlich Komponenten, die ein konkretes graphisches Objekt darstellen, das keine weiteren Unterkomponenten enthalten kann. Solche Komponenten sind z.B.

`JButton, JTextField, JTable` und `JComBox`.

Diese Komponenten lassen sich über ihre Konstruktoren instanziieren, um sie dann einer Zwischenkomponenten über deren Methode `add` als Unterkomponente hinzuzufügen. Sind alle gewünschten graphischen Objekte einer Zwischenkomponente hinzugefügt worden, so kann auf der zugehörigen top-level Komponenten die Methode `pack` aufgerufen wurden. Diese berechnet die notwendige Größe und das Layout des Fensters, welches schließlich sichtbar gemacht wird.

Das folgende Programm erzeugt ein Fenster mit einer Textfläche.

```
1  import javax.swing.*;
2
3  class JT {
4    public static void main(String [] _){
5      JFrame frame = new JFrame();
6      JTextArea textArea = new JTextArea();
7      textArea.setText("hallo da draußen");
8      frame.add(textArea);
9      frame.pack();
10     frame.setVisible(true);
11   }
12 }
```

Listing 5.3: JT

In der Regel wird man die GUI-Komponente, die man schreibt nicht nacheinander in der Hauptmethode definieren, sondern es wird ein eigenes Objekt definiert, dass die GUI Komponente darstellt. Hierzu leitet man eine spezifische GUI-Klasse von der Klasse `JPanel` ab und fügt bereits im Konstruktor die entsprechenden Unterkomponenten hinzu.

Die folgende Klasse definiert eine Komponente, die einen Knopf und eine Textfläche enthält. In der Hauptmethode wird das Objekt instantiiert:

```
1  package name.panitz.simpleGui;
2  import javax.swing.*;
3
4  class JTB extends JPanel {
5    JTextArea textArea = new JTextArea();
6    JButton button = new JButton("ein knopf");
7
8    public JTB(){
9      textArea.setText("hallo da draußen");
10     add(textArea);
11     add(button);
12   }
13
14   public void showInFrame(){
15     JFrame f = new JFrame();
16     f.add(this);
17     f.pack();
18     f.setVisible(true);
19   }
20
21  public static void main(String [] _){
22     new JTB().showInFrame();
23   }
24 }
```

Listing 5.4: JTB.java

5.2 Gruppierungen

Bisher haben wir Unterkomponenten weitere Komponenten mit der Methode `add` hinzugefügt, ohne uns Gedanken über die Plazierung der Komponenten zu machen. Wir haben einfach auf das Standardverhalten zur Plazierung von Komponenten vertraut. Ob die Komponenten schließlich nebeneinander, übereinander oder irgendwie anders Gruppiert im Fenster erschienen, haben wir nicht spezifiziert.

Um das Layout von graphischen Komponenten zu steuern, steht das Konzept der sogenannten *Layout-Manager* zur Verfügung. Ein Layout-Manager ist ein Objekt, das einer Komponente hinzugefügt wird. Der Layout-Manager steuert dann, in welcher Weise die Unterkomponenten gruppiert werden.

`LayoutManager` ist eine Schnittstelle. Es gibt mehrere Implementierungen dieser Schnittstelle. Wir werden in den nächsten Abschnitten drei davon kennenlernen. Es steht einem natürlich frei, eigene Layout-Manager durch Implementierung dieser Schnittstelle zu schreiben. Es wird aber davon abgeraten, weil dieses notorisch schwierig ist und die in Java bereits vorhandenen Layout-Manager bereits sehr mächtig und ausdrucksstark sind.

Zum Hinzufügen eines Layout-Manager gibt es die Methode `setLayout`.

5.2.1 Flow Layout

Der vielleicht einfachste Layout-Manager nennt sich `FlowLayout`. Hier werden die Unterkomponenten einfach der Reihe nach in einer Zeile angeordnet. Erst wenn das Fenster zu schmal hierzu ist, werden weitere Komponenten in eine neue Zeile gruppiert.

Die folgende Klasse definiert ein Fenster, dessen Layout über ein Objekt der Klasse `FlowLayout` gesteuert wird. Dem Fenster werden fünf Knöpfe hinzugefügt:

```
1  package name.panitz.gui.layoutTest;
2
3  import java.awt.*;
4  import javax.swing.*;
5
6  class FlowLayoutTest extends JFrame {
7
8    public FlowLayoutTest(){
9      JPanel pane = new JPanel();
10     pane.setLayout(new FlowLayout());
11     pane.add(new JButton("eins"));
12     pane.add(new JButton("zwei"));
13     pane.add(new JButton("drei (ein langer Knopf)"));
14     pane.add(new JButton("vier"));
15     pane.add(new JButton("fuenf"));
16     add(pane);
17     pack();
18     setVisible(true);
19   }
20
21   public static void main(String [] _){new FlowLayoutTest();}
22 }
```

Listing 5.5: FlowLayoutTest.java

Das Fenster hat die optische Ausprägung aus Abbildung 4.3.1.

Abbildung 5.1: Anordnung durch das Flow Layout.

Verändert man mit der Maus die Fenstergröße, macht es z.B. schmal und hoch, so werden die Knöpfe nicht mehr nebeneinander sonder übereinander angeordnet.

5.2.2 Border Layout

Die Klasse `BorderLayout` definiert einen Layout Manager, der fünf feste Positionen kennt: eine Zentralposition, und jeweils links/rechts und oberhalb/unterhalb der Zentralposition eine Position für Unterkomponenten. Die Methode `add` kann in diesem Layout auch noch mit einem zweitem Argument aufgerufen werden, das eine dieser fünf Positionen angibt. Hierzu bedient man sich der konstanten Felder der Klasse `BorderLayout`.

In dieser Klasse wird die Klasse `BorderLayout` zur Steuerung des Layoout benutzt. Die fünf Knöpfe werden an jeweils eine der fünf Positionen hinzugefügt:

```
1  package name.panitz.gui.layoutTest;
2
3  import java.awt.*;
4  import javax.swing.*;
5
6  class BorderLayoutTest extends JFrame {
7
8    public BorderLayoutTest(){
9      JPanel pane = new JPanel();
10     pane.setLayout(new BorderLayout());
11     pane.add(new JButton("eins"),BorderLayout.NORTH);
12     pane.add(new JButton("zwei"),BorderLayout.SOUTH);
13     pane.add(new JButton("drei (ein langer Knopf)")
14                                  ,BorderLayout.CENTER);
15     pane.add(new JButton("vier"),BorderLayout.WEST);
16     pane.add(new JButton("fuenf"),BorderLayout.EAST);
17     add(pane);
18     pack();
19     setVisible(true);
20   }
21
22   public static void main(String [] _){new BorderLayoutTest();}
23 }
```

Listing 5.6: BorderLayoutTest.java

Die Klasse erzeugt das Fenster aus Abbildung 4.3.1.

Das Layout ändert sich nicht, wenn man mit der Maus die Größe und das Format des Fensters verändert.

5.2.3 Grid Layout

Die Klasse `GridLayout` ordnet die Unterkomponenten tabellarisch an. Jede Komponente wird dabei gleich groß ausgerichtet. Die Größe richtet sich also nach dem größten Element.

Folgende Klasse benutzt ein Grid-Layout mit zwei Zeilen zu je drei Spalten.

Abbildung 5.2: Anordnung über das Border Layout

```
1  package name.panitz.gui.layoutTest;
2
3  import java.awt.*;
4  import javax.swing.*;
5
6  class GridLayoutTest extends JFrame {
7    public GridLayoutTest(){
8      JPanel pane = new JPanel();
9      pane.setLayout(new GridLayout(2,3));
10     pane.add(new JButton("eins"));
11     pane.add(new JButton("zwei"));
12     pane.add(new JButton("drei (ein langer Knopf)"));
13     pane.add(new JButton("vier"));
14     pane.add(new JButton("fünf"));
15     add(pane);
16     pack();
17     setVisible(true);
18   }
19
20   public static void main(String [] _){new GridLayoutTest();}
21 }
```

Listing 5.7: GridLayoutTest.java

Folgendes Fensters wird durch dieses Programm geöffnet.

Abbildung 5.3: Anordnung im Grid Layout

Auch hier ändert sich das Layout nicht, wenn man mit der Maus die Größe und das Format des Fensters verändert.

5.3 Eigene GUI-Komponenten

Bisher graphische Komponenten unter Verwendung der fertigen GUI-Komponente aus der Swing-Bibliothek zusammengesetzt. Oft will man graphische Komponenten schreiben, für die es keine fertige GUI-Komponente in der Swing-Bibliothek gibt. Wir müssen wir eine entsprechende Komponente selbst schreiben.

Um eine eigene GUI-Komponente zu schreiben, schreibt man eine Klasse, die von der GUI-Klasse ableitet. Dieses haben wir bereits in den letzten Beispielen getan, indem wir von der Klasse `JFrame` abgelitten haben. Die dort geschriebenen Unterklassen der Klasse `JFrame` zeichneten sich dadurch aus, daß sie eine Menge von graphischen Objekten (Knöpfe, Textfelder…) in einer Komponente zusammengefasst haben. In diesem Abschnitt werden wir eine neue Komponente definieren, die keine der bestehenden fertigen Komponenten benutzt, sondern selbst alles zeichnet, was zu ihrer Darstellung notwendig ist.

Hierzu betrachten wir eine der entscheidenen Methoden der Klasse `JComponent`, die Methode `paintComponent`. In dieser Methode wird festgelegt, was zu zeichnen ist, wenn die graphische Komponente darzustellen ist. Die Methode `paintComponent` hat folgende Signatur:

```
1 public void paintComponent(java.awt.Graphics g)
```

Java ruft diese Methode immer auf, wenn die graphische Komponente aus irgendeinen Grund zu zeichnen ist. Dabei bekommt die Methode das Objekt übergeben, auf dem gezeichnet wird. Dieses Objekt ist vom Typ `java.awt.Graphics`. Es stellt ein zweidimensionales Koordinatensystem dar, in dem zweidimensionale Graphiken gezeichnet werden können. Der Nullpunkt dieses Koordinatensystems ist oben links und nicht unten links, wie wir es vielleicht aus der Mathematik erwartet hätten.

In der Klasse `Graphics` sind eine Reihe von Methoden definiert, die es erlauben graphische Objekte zu zeichnen. Es gibt Methoden zum Zeichnen von Geraden, Vierecken, Ovalen, beliebigen Polygonzügen, Texten etc.

Wollen wir eine eigene graphische Komponente definieren, so können wir die Methode `paintComponent` überschreiben und auf dem übergebenen Objekt des Typs `Graphics` entsprechende Methoden zum Zeichnen aufrufen. Um eine eigene graphische Komponente zu definieren, wird empfohlen die Klasse `JPanel` zu erweitern und in ihr die Methode `paintComponent` zu überschreiben.

Folgende Klasse definiert eine neue graphische Komponente, die zwei Linien, einen Text, ein Rechteck, ein Oval und ein gefülltes Kreissegment enthält.

```
1 package name.panitz.gui.graphicsTest;
2
3 import javax.swing.JPanel;
```

```
4  import javax.swing.JFrame;
5  import java.awt.Graphics;
6
7  class SimpleGraphics extends JPanel{
8    public void paintComponent(Graphics g){
9      g.drawLine(0,0,100,200);
10     g.drawLine(0,50,100,50);
11     g.drawString("hallo",10,20);
12     g.drawRect(10, 10, 60,130);
13     g.drawOval( 50,  100, 30, 80);
14     g.fillArc(-20, 150, 80, 80, 0, 50);
15   }
16 }
```

Listing 5.8: SimpleGraphics.java

Diese Komponente können wir wie jede andere Komponente auch einem Fenster hinzufügen, so daß sie auf dem Bildschirm angezeigt werden kann.

```
1  package name.panitz.gui.graphicsTest;
2
3  import javax.swing.JFrame;
4
5  class UseSimpleGraphics {
6    public static void main(String [] args){
7      JFrame frame = new JFrame();
8      frame.getContentPane().add(new SimpleGraphics());
9
10     frame.pack();
11     frame.setVisible(true);
12   }
13 }
```

Listing 5.9: UseSimpleGraphics.java

Ärgerlich in unserem letzten Beispiel war, daß Java zunächst ein zu kleines Fenster für unsere Komponente geöffnet hat, und wir dieses Fenster mit Maus erst größer ziehen mußten. Die Klasse `JComponent` enthält Methoden, in denen die Objekte angeben können, welches ihre bevorzugte Größe bei ihrere Darstellung ist. Wenn wir diese Methode überschreiben, so daß sie eine Dimension zurückgibt, in der das ganze zu zeichnende Bild passt, so wird von Java auch ein entsprechend großes Fenster geöffnet. Wir fügen der Klasse `SimpleGraphics` folgende zusätzliche Methode hinzu.

```
1    public java.awt.Dimension getPreferredSize() {
2      return new java.awt.Dimension(100,200);
3    }
```

Jetzt öffnet Java ein Fenster, in dem das ganze Bild dargestellt werden kann.

5.3.1 Fraktale

Um noch ein wenig mit Farben zu spielen, zeichnen wir in diesem Abschnitt die berühmten Apfelmännchen. Apfelmännchen werden definiert über eine Funktion auf komplexen Zahlen. Die aus der Mathematik bekannten komplexen Zahlen sind Zahlen mit zwei reellen Zahlen als Bestandteil, den sogenannten Imaginärteil und den sogenannten Realteil. Wir schreiben zunächst eine rudimentäre Klasse zur Darstellung von komplexen Zahlen:

```
1 package name.panitz.crempel.tool.apfel;
2
3 public class Complex{
```

Listing 5.10: Complex.java

Diese Klasse braucht zwei Felder um Real- und Imaginärteil zu speichern:

```
4   public double re;
5   public double im;
```

Listing 5.11: Complex.java

Ein naheliegender Konstruktor für komplexe Zahlen füllt diese beiden Felder.

```
6  public Complex(double re,double im){
7   this.re=re;this.im=im;
8 }
```

Listing 5.12: Complex.java

Im Mathematikbuch schauen wir nach, wie Addition und Multiplikation für komplexe Zahlen definiert sind, und schreiben entsprechende Methoden:

```
9   public Complex add(Complex other){
10    return new Complex(re+other.re,im+other.im);
11  }
12
13  public Complex mult(Complex other){
14    return new Complex
15      (re*other.re-im*other.im,re*other.im+im*other.re);
16  }
```

Listing 5.13: Complex.java

Zusätzlich finden wir in Mathematik noch die Definition der Norm einer komplexen Zahl und setzen auch diese Definition in eine Methode um. Zum Quadrat des Realteils wird das Quadrat des Imaginärteils addiert.

```
17   public double norm(){return re*re+im*im;}
18 }
```

Listing 5.14: Complex.java

Soweit komplexe Zahlen, wie wir sie für Apfelmännchen brauchen.

Grundlage zum Zeichnen von Apfelmännchen ist folgende Iterationsgleichung auf komplexen Zahlen: $z_{n+1} = z_n^2 + c$. Wobei z_0 die komplexe Zahl $0+0i$ mit dem Real- und Imaginärteil 0 ist.

Zum Zeichnen der Apfelmännchen wird ein Koordinatensystem so interpretiert, daß die Achsen jeweils Real- und Imaginärteil von komplexen Zahlen darstellen. Jeder Punkt in diesem Koordinatensystem steht jetzt für die Konstante c in obiger Gleichung. Nun wir geprüft ob und für welches n die Norm von z_n größer eines bestimmten Schwellwertes ist. Je nach der Größe von n wird der Punkt im Koordinatensystem mit einer anderen Farbe eingefärbt.

Mit diesem Wissen können wir nun versuchen die Apfelmännchen zu zeichnen. Wir müssen nur geeignete Werte für die einzelnen Parameter finden. Wir schreiben eine eigene Klasse für das graphische Objekt, in dem ein Apfelmännchen gezeichnet wird. Wir deklarieren die Imports der benötigten Klassen:

```
1  package name.panitz.crempel.tool.apfel;
2
3  import java.awt.Graphics;
4  import java.awt.Color;
5  import java.awt.Dimension;
6  import javax.swing.JFrame;
7  import javax.swing.JPanel;
8
9  public class Apfelmaennchen extends JPanel {
```

Listing 5.15: Apfelmaennchen.java

Als erstes deklarieren wir Konstanten für die Größe des Apfelmännchens.

```
10   final int width = 480;
11   final int height = 430;
```

Listing 5.16: Apfelmaennchen.java

Eine weitere wichtige Konstante ist der Faktor, der angibt, welcher reellen Zahl ein Pixel entspricht:

```
12   double zelle=0.00625;
```

Listing 5.17: Apfelmaennchen.java

Eine weitere Konstanten legt die Farbe fest, mit der die Punkte, die nicht über einen bestimmten Schwellwert konvergieren, eingefärbt werden sollen:

```
13   final Color colAppleman = new Color(0,129,190);
```

Listing 5.18: Apfelmaennchen.java

Weitere Konstanten legen fest welche komplexe Zahl der Nullpunkt unseres `Graphics`-Objekts darstellt.

```
14   double startX = -2;
15   double startY = -1.35;
```

Listing 5.19: Apfelmaennchen.java

Weitere Konstanten sind der Schwellwert und die maximale Rekursionstiefe n , für die wir jeweils z_n berechnen:

```
16   final int recDepth = 50;
17   final int schwellwert = 4;
```

Listing 5.20: Apfelmaennchen.java

Die wichtigste Methode berechnet die Werte für die Gleichung $z_{n+1} = z_n^2 + c$. Der Eingabeparameter ist die komplexe Zahl c . Das Ergebnis dieser Methode ist das n , für das z_n größer als der Schwellwert ist:

```
18   //C-Werte checken nach zn+1 = zn*zn + c,
19   public int checkC(Complex c) {
20     Complex zn = new Complex(0,0);
21
22     for (int n=0;n<recDepth;n=n+1) {
23       final Complex znp1 = zn.mult(zn).add(c);
24       if (znp1.norm() > schwellwert) return n;
25       zn=znp1;
26     }
27     return recDepth;
28   }
```

Listing 5.21: Apfelmaennchen.java

Jetzt gehen wir zum Zeichnen jedes Pixel unseres `Graphics`-Objekts durch, berechnen welche komplexe Zahl an dieser Stelle steht und benutzen dann die Methode *checkC* , um zu berechnen ob und nach wieviel Iterationen die Norm von z_n größer als der Schwellwert wird. Abhängig von dieser Zahl, färben wir den Punkt mit einer Farbe ein.

```
29   public void paint(Graphics g) {
30     for (int y=0;y<height;y=y+1) {
31       for (int x=0;x<width;x=x+1) {
32
33         final Complex current
34           =new Complex(startX+x*zelle,startY+y*zelle);
35
36         final int iterationenC = checkC(current);
37
38         paintColorPoint(x,y,iterationenC,g);
39       }
40     }
41   }
```

Listing 5.22: Apfelmaennchen.java

Zur Auswahl der Farbe benutzen wir folgende kleine Methode, die Abhängig von ihrem Parameter `it` an der Stelle `(x,y)` einen Punkt in einer bestimmten Farbe zeichnet.

```
42   private void paintColorPoint
43                 (int x,int y,int it,Graphics g){
44     final Color col
45       = it==recDepth
46         ?colAppleman
47         :new Color(255-5*it%1,255-it%5*30,255-it%5* 50);
48     g.setColor(col);
49     g.drawLine(x,y,x,y);
50   }
```

Listing 5.23: Apfelmaennchen.java

Schließlich können wir noch die Größe festlegen und das Ganze in einer Hauptmethode starten:

```
51   public Dimension getPreferredSize(){
52     return new Dimension(width,height);
53   }
54
55   public static void main(String [] args){
56     JFrame f = new JFrame();
57     f.getContentPane().add(new Apfelmaennchen());
58     f.pack();
59     f.setVisible(true);
60   }
61 }
```

Listing 5.24: Apfelmaennchen.java

Das Programm ergibt das Bild aus Abbildung 5.4.

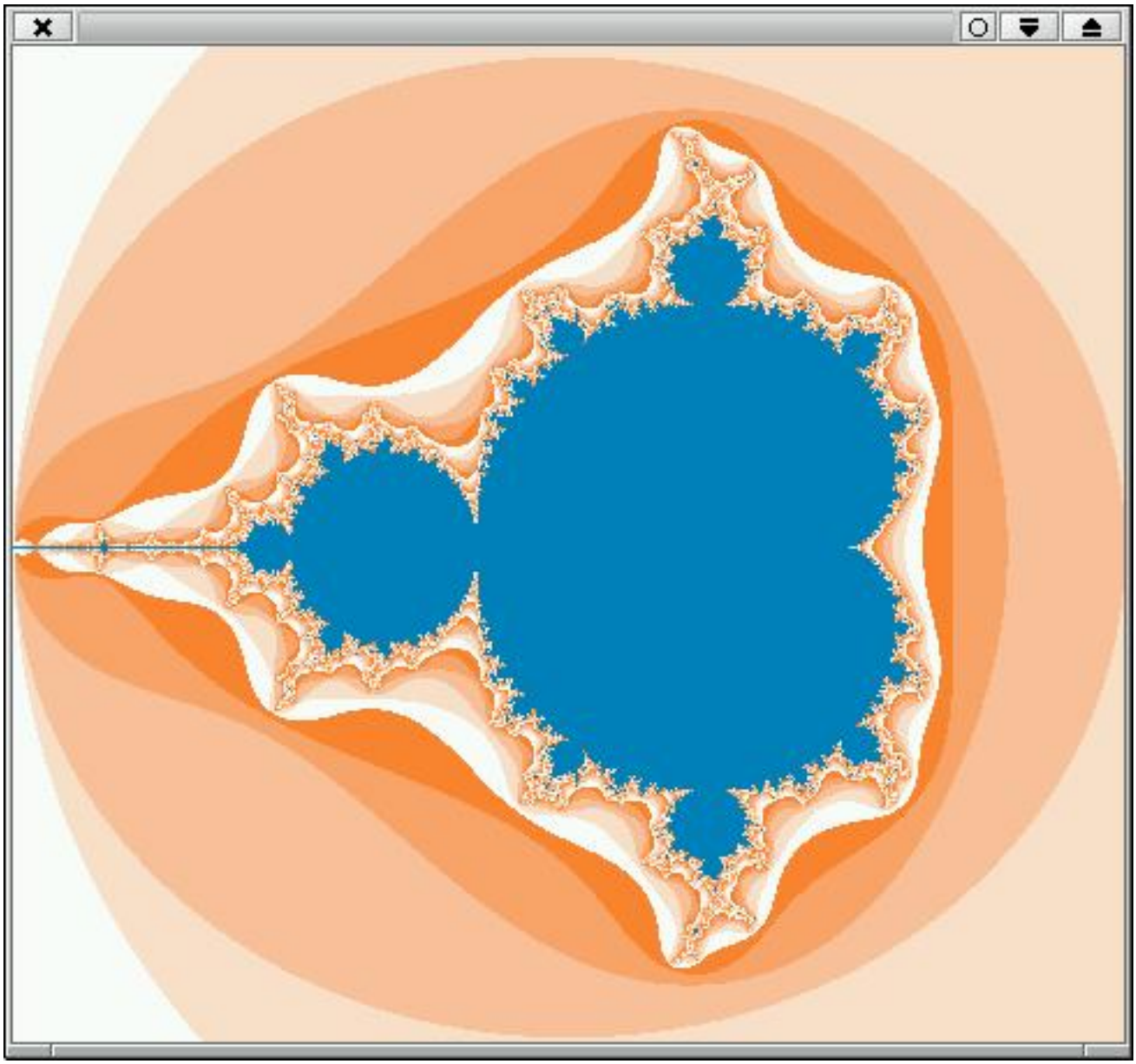

Abbildung 5.4: Das berühmte Apfelmännchen.

5.4 Reaktion auf Ereignisse

Um den graphischen Komponenten eine Funktionalität hinzuzufügen, kennt Java das Konzept der Ereignisbehandlung. Graphische Objekte sollen in der Regel auf bestimmte Ereignisse auf eine definierte Weise reagieren. Solche Ereignisse könne Mausbewegungen, Mausklicks, Ereignisse an einem Fenster, wie das Schließen des Fensters oder etwa Eingaben auf der Tastatur sein. Für die verschiedenen Arten von Ereignissen sind im Paket `java.awt.event` Schnittstellen definiert. In diesen

Schnittstellen stehen Methoden, in denen die Reaktion auf bestimmte Ereignisse definiert werden kann. So gibt es z.B. eine Schnittstelle `Mouselistener`, in der Methoden für verschiedene Ereignisse auf den Mausknöpfen bereitstehen.

Soll einer bestimmten graphischen Komponente eine bestimmte Reaktion auf bestimmte Ereignisse zugefügt werden, so ist die entsprechende Schnittstelle mit Methoden für das anvisierte Ereignis ausgeguckt und implementiert werden. Ein Objekt dieser Implementierung kann dann der graphischen Komponente mit einer entsprechenden Methode hinzugefügt werden.

5.4.1 Der `ActionListener`

Das allgemeinste Ereignis ist ein `ActionEvent`. Die entsprechende Schnittstelle `ActionListener` enthält nur eine Methode, die auszuführen ist, wenn eine Aktion aufgetreten ist. Dieses Ereignis wird von einem Knopf-Objekt der Klasse `JButton` ausgelöst, wenn ein benutzer auf den Knopf mit der Maus klickt.

Wir implementieren die Schnittstelle `ActionListener` so, dass in einem internen Zähler vermerkt wird, wie oft ein Ereignis aufgetreten ist. Bei jedem Auftreten des Ereignisses wird die entsprechende Zahl auf einer Textfläche gesetzt:

```
1  import java.awt.event.*;
2  import javax.swing.text.*;
3
4  class CountActionListener implements ActionListener {
5    JTextComponent textArea;
6    int count;
7
8    CountActionListener(JTextComponent textArea){
9      this.textArea=textArea;
10   }
11
12   public void actionPerformed(ActionEvent e) {
13     count = count+1;
14     textArea.setText(""+count);
15   }
16 }
```

Listing 5.25: CountActionListener

In einer zweiten Klasse definieren wir eine Fensterkomponente mit zwei atomaren Komponenten: einen Knopf und eine Textfläche. Dem Knopf fügen wir die oben geschriebene Ereignisbehandlung hinzu.

```
1  import java.awt.*;
2  import javax.swing.*;
3  import javax.swing.text.*;
4
```

```
5  class Count extends JFrame {
6
7    public Count(){
8      JTextComponent textArea = new JTextField(8);
9      JButton button = new JButton("click");
10     JPanel pane = new JPanel();
11     pane.add(button);
12     pane.add(textArea);
13
14     add(pane);
15     button.addActionListener(new CountActionListerner(textArea));
16     pack();
17   }
18
19   public static void main(String [] _){
20     JFrame f = new Count();
21     f.setVisible(true);
22   }
23 }
```

Listing 5.26: Count

Wir erhalten ein Fenster, in dem die Anzahl der Mausklicks auf dem Knopf im Textfeld angezeigt wird.

5.4.2 Innere und Anonyme Klassen

Innere Klassen

Eine innere Klasse wird geschrieben wie jede andere Klasse auch, nur dass sie eben im Rumpf einer äußeren Klasse auftauchen kann. Die innere Klasse hat das Privileg auf die Eigenschaften der äußeren Klasse zuzugreifen, sogar auf die als privat markierten Eigenschaften. Das Attribut privat soll lediglich verhindern, dass eine Eigenschaft von außerhalb der Klasse benutzt wird. Innere Klasse befinden sich aber innerhalb der Klasse.

Unser erstes GUI mit einer Funktionalität lässt sich jetzt mit Hilfe einer inneren Klasse in einer Quelltext-Datei schreiben. Das Feld `counter`, das wir in der vorherigen Implementierung als privates Feld der Klass `CounterListener` definiert hatten, haben wir hier als Feld der GUI-Klasse modelliert. Trotzdem kann die Klasse `CounterListener` weiterhin darauf zugreifen. Ebenso brauch die Textfläche nicht der Klasse `CounterListener` im Konstruktor übergeben werden. Als innere Klasse kann in `CounterListener` auf dieses Feld der äußeren Klasse zugegriffen werden.

```
1 package name.panitz.simpleGui;
2 import javax.swing.*;
3 import java.awt.event.*;
4
```

```
5  class InnerCounter extends JTB {
6    private int counter = 0;
7
8    class CounterListener implements ActionListener{
9      public void actionPerformed(ActionEvent _){
10       counter = counter+1;
11       textArea.setText(counter+"");
12     }
13   }
14
15   public InnerCounter(){
16     button.addActionListener(new CounterListener());
17   }
18
19   public static void main(String [] _){new InnerCounter().showInFrame
       ();}
20 }
```

Listing 5.27: InnerCounter.java

Tatsächlich ist die Implementierung kürzer und etwas übersichtlicher geworden.

Beim Übersetzen einer Klasse mit inneren Klassen, erzeugt der Javaübersetzer für jede innere Klasse eine eigene Klassendatei:

```
sep@linux:~/fh/ooseAI/classes/name/panitz/simpleGui> ll *.class
-rw-r--r--   1 sep    users     1082 2014-03-29 11:36 InnerCounter$CounterListener.class
-rw-r--r--    1 sep      users         892 2014-03-29 11:36 InnerCounter.class
```

Der Javaübersetzer schreibt intern den Code um in eine Menge von Klassen ohne innere Klassendefinition und erzeugt für diese den entsprechenden Code. Für die innere Klasse generiert der Javaübersetzer einen Namen, der sich aus äußeren und inneren Klassenamen durch ein Dollarzeichen getrennt zusammensetzt.

Anonyme Klassen

Im letzten Abschnitt hatten wir bereits das Beispiel einer inneren Klasse, für die wir genau einmal ein Objekt erzeugen. In diesem Fall wäre es eigentlich unnötig für eine solche Klasse einen Namen zu erfinden, wenn man an genau dieser einen Stelle, an der das Objekt erzeugt wird, die entsprechende Klasse spezifizieren könnte. Genau hierzu dienen anonyme Klassen in Java. Sie ermöglichen, Klassen ohne Namen zu instanziieren. Hierzu ist nach dem Schlüsselwort `new` anzugeben, von welcher Oberklasse namenlose Klasse ableiten soll, oder welche Schnittstelle mit der namenlosen Klasse implementiert werden soll. Dann folgt nach dem leeren Klammerpaar für dem Konstruktoraufruf in geschweiften Klammern der Rumpf der namenlosen Klasse.

Wir schreiben ein drittes Mal die Klasse `Counter`. Diesmal wird statt der nur einmal instanziierten inneren Klasse eine anonyme Implementierung der Schnittstelle `ActionListener` Instanziiert.

```
1  package name.panitz.simpleGui;
2  import javax.swing.*;
3  import java.awt.event.*;
4
5  class AnonymousCounter extends JTB {
6    private int counter = 0;
7
8    public AnonymousCounter(){
9      button.addActionListener(
10       new ActionListener(){
11         public void actionPerformed(ActionEvent _){
12           counter = counter+1;
13           textArea.setText(counter+"");
14         }
15       });
16   }
17
18   public static void main(String [] _){new AnonymousCounter().
       showInFrame();}
19 }
```

Listing 5.28: AnonymousCounter.java

Auch für anonyme Klassen generiert der Javaübersetzer eigene Klassendateien. Mangels eines Names, numeriert der Javaübersetzer hierbei die inneren Klassen einfach durch.

```
sep@linux:~/fh/prog2/examples/classes/name/panitz/simpleGui> ll *.class
-rw-r--r--    1 sep      users       1106 2004-03-29 11:59 AnonymousCounter$1.class
-rw-r--r--    1 sep      users        887 2004-03-29 11:59 AnonymousCounter.class
sep@linux:~/fh/prog2/examples/classes/name/panitz/simpleGui>
```

5.4.3 Lambda Ausdrücke

Seit Java 8 im Jahre 2014 gibt es eine noch kompaktere Möglichkeit, um einem Knopf die gewünschte Aktion hinzuzufügen. Die Schnittstelle `ActionListener` enthält nur eine einzige Methode. Solche Schnittstelle werden nun als funktionale Schnittstellen bezeichnet, weil sie nur eine Funktion in Form einer Methode enthalten.

Mit Java 8 wurde eine neue Art von Ausdrücken eingeführt. Diese nennen sich Lambda-Ausdrücke. Sie bestehen aus einer Parameterliste in runden Klammern, gefolgt von einem Pfeil, der durch das Minussymbol und das Größersymbol gebildet wird, also `->`. Nach dem Pfeil folgt ein Methodenrumpf oder direkt ein Ausdruck, der zu einem Ergebnis auswertet.

Ein solcher Lambda-Ausdruck ist die Kurzschreibweise für eine Implementierung einer funktionalen Schnittstelle. Es wird also nicht mehr angegeben, welche

Schnittstelle implementiert wird, auch nicht mehr, wie die Methode heißt, die implementiert wird, sondern nur noch die Parameterliste und der Methodenrumpf. Alles andere erkennt der Javaübersetzer aus dem Kontext.

Damit wird die kleine Counteranwendung noch kompakter ausgedrückt:

```
1  package name.panitz.simpleGui;
2  import javax.swing.*;
3  import java.awt.event.*;
4
5  class LambdaCounter extends JTB {
6    private int counter = 0;
7
8    public LambdaCounter(){
9      button.addActionListener( (ev)->{
10           counter = counter+1;
11           textArea.setText(counter+"");
12       });
13   }
14
15   public static void main(String [] _){new LambdaCounter().showInFrame
       ();}
16 }
```

Listing 5.29: LambdaCounter.java

5.4.4 Mausereignisse

Die Schnittstelle `ActionListener` ist dazu geeignet, die Reaktionen einfachsten Ereignisse zu programmieren, den Drücken eines Knopfes.

Ein in modernen graphischen Oberflächen häufigst benutztes Eingabemedium ist die Maus. Zwei verschiedene Ereignisarten sind für die Maus relevant:

- Mausereignisse, die sich auf das Drücken, Freilassen oder Klicken auf einen der Mausknöpfe bezieht. Hierfür gibt es eine Schnittstelle zur Behandlung solcher Ereignisse: `MouseListener`.
- Mausereignisse, die sich auf das Bewegen der Maus beziehen. Die Behandlung solcher Ereignisse kann über eine Implementierung der Schnittstelle `MouseMotionListener` spezifiziert werden.

Entsprechend gibt es für graphische Komponenten Methoden, um solche Mausereignisbehandler der Komponente hinzuzufügen: `addMouseListener` und `addMouseMotionListener`.

Um die Arbeit mit Ereignisbehandlern zu vereinfachen, gibt es für die entsprechnden Schnittstellen im Paket `java.awt.event` prototypische Implementierungen, in denen die Methoden der Schnittstelle so implementiert sind, daß ohne Aktion

auf die entsprechenden Ereignisse reagiert wird. Diese prototypischen Implementierung sind Klassen, deren Namen mit *Adapter* enden. So gibt es zur Schnittstelle `MouseListener` die implementierende Klasse `MouseAdapter`. Will man eine bestimmte Mausbehandlung programmieren, reicht es aus, diesen Adapter zu erweitern und nur die Methoden zu überschreiben, für die bestimmte Aktionen vorgesehen sind. Es erübrigt sich dann für alle sechs Methoden der Schnittstelle `MouseListener` Implementierungen vorzusehen.

Wir erweitern die Klasse `Apfelmaennchen` um eine Mausbehandlung. Der mit gedrückter Maus markierte Bereich soll vergrößert in dem Fenster dargestellt werden.

```
1  package name.panitz.crempel.tool.apfel;
2
3  import java.awt.Graphics;
4  import java.awt.event.*;
5  import javax.swing.JFrame;
6
7  public class ApfelWithMouse extends Apfelmaennchen{
8    public ApfelWithMouse(){
```

Listing 5.30: ApfelWithMouse.java

Im Konstruktor fügen wir der Komponente eine Mausbehandlung hinzu. Der Mausbehandler merkt sich die Koordinaten, an denen die Maus gedrückt wird und berechnet beim Loslassen des Mausknopfes den neuen darzustellenden Zahlenbereich:

```
9      addMouseListener(new MouseAdapter(){
10       int mouseStartX=0;
11       int mouseStartY=0;
12
13       public void mousePressed(MouseEvent e) {
14         mouseStartX=e.getX();
15         mouseStartY=e.getY();
16       }
17
18       public void mouseReleased(MouseEvent e) {
19         int endX = e.getX();
20         int endY = e.getY();
21         startX = startX+(mouseStartX*zelle);
22         startY = startY+(mouseStartY*zelle);
23         zelle  = zelle*(endX-mouseStartX)/width;
24         repaint();
25       }
26     });
27   }
```

Listing 5.31: ApfelWithMouse.java

Auch für diese Klasse sehen wir eine kleine Startmethode vor:

```
28   public static void main(String [] _){
29     JFrame f = new JFrame();
30     f.getContentPane().add(new ApfelWithMouse());
31     f.pack();
32     f.setVisible(true);
33   }
34 }
```

Listing 5.32: ApfelWithMouse.java

5.4.5 Fensterereignisse

Auch für Fenster in einer graphischen Benutzeroberfläche existieren eine Reihe von Ereignissen. Das Fenster kann minimiert oder maximiert werden, es kann das aktive Fenster oder im Hintergrund sein und es kann schließlich auch geschlossen werden. Um die Reaktion auf solche Ereignisse zu spezifizieren existiert die Schnittstelle `WindowListener` mit entsprechender prototypischer Adapterklasse `WindowAdapter`. Die Objekte der Fensterereignisbehandlung können mit der Methode `addWindowListener` Fensterkomponenten hinzugefügt werden.

In den bisher vorgestellten Programmen wird Java nicht beendet, wenn das einzige Fenster der Anwendung geschlossen wurde. Man kann an der Konsole sehen, dass der Javainterpreter weiterhin aktiv ist. Das liegt daran, dass wir bisher noch nicht spezifiziert haben, wie die Fensterkomponenten auf das Ereignis des Schließens des Fensters reagieren sollen. Dieses kann mit einem Objekt, das `WindowListener` implementiert in der Methode `windowClosing` spezifiziert werden. Wir schreiben hier eine Version des Apfelmännchenprogramms, in dem das Schließen des Fensters den Abbruch des gesamten Programms bewirkt.

```
1  package name.panitz.crempel.tool.apfel;
2
3  import javax.swing.JFrame;
4  import java.awt.event.*;
5
6  public class ClosingApfelFrame {
7    public static void main(String [] args){
8      JFrame f = new JFrame();
9      f.add(new ApfelWithMouse());
10     f.addWindowListener(new WindowAdapter(){
11       public void windowClosing(WindowEvent e) {
12         System.exit(0);
13       }
14     });
15     f.pack();
16     f.setVisible(true);
17   }
18 }
```

Listing 5.33: ClosingApfelFrame.java

5.5 Zeitgesteuerte Ereignisse

Um zeitlich immer wiederkehrende Ereignisse in GUIs zu programmieren gibt in Swing eine Hilfsklasse `Timer`. Objekte dieser Klasse können so instanziiert werden, dass sie in bestimmten Zeitabständen Ereignisse auslösen. Der `Timer` ist also so etwas wie ein Ereignisgenerator. Zusätzlich gibt man einem `Timer`-Objekt auch einen `ActionListener` mit, der spezifiziert, wie auf diese in Zeitintervallen auftretenden Ereignisse reagiert werden soll.

Folgende Klasse implementiert eine simple Uhr. In einem `JLabel` wird die aktuelle Zeit angegeben. Die Komponente wird einem `Timer` übergeben, der jede Sekunde eine neues Ereignis erzeugt. Diese Ereignisse sorgen dafür, dass die Zeit im Label aktualisiert wird.

```
1 package name.panitz.oose.swing.examples;
2
3 import javax.swing.*;
4 import java.util.Date;
5 import java.awt.event.*;
```

Listing 5.34: Uhr.java

Die Klasse Uhr ist nicht nur ein `JPanel`, in dem ein `JLabel` benutzt wird, Datum und Uhrzeit anzuzeigen, sondern implementiert gleichfalls auch einen `ActionListener`.

```
6 public class Uhr extends JPanel implements ActionListener{
```

Listing 5.35: Uhr.java

Zunächst sehen wir das Datumsfeld für diese Komponente vor:

```
7   JLabel l = new JLabel(new Date()+"");
```

Listing 5.36: Uhr.java

Im Konstruktor erzeugen wir ein Objekt vom Typ `Timer`. Dieses Objekt soll alle Sekunde (alle 1000 Millisekunden) ein Ereignis erzeugen. Dem `Timer` wird das gerade im Konstruktor erzeugte Objekt vom Typ `Uhr` übergeben, das, da es ja einen `ActionListener` implementiert, auf diese Ereignisse reagieren soll.

```
8   public Uhr (){
9     new Timer(1000,this).start();
10    add(l);
11  }
```

Listing 5.37: Uhr.java

Um die Schnittstelle `ActionListener` korrekt zu implementieren, muss die Methode `actionPerformed` implementiert werden. In dieser setzen wir jeweils Datum und Uhrzeit mit dem aktuellen Wert neu ins Label.

```
12  public void actionPerformed(ActionEvent _){
13    l.setText(""+new Date());
14  }
```

Listing 5.38: Uhr.java

Und natürlich sehen wir zum Testen eine kleine Hauptmethode vor, die die Uhr in einem Fensterrahmen anzeigt.

```
15  public static void main(String [] _){
16    JFrame f = new JFrame();
17    f.getContentPane().add(new Uhr());
18    f.pack();
19    f.setVisible(true);
20  }
21}
```

Listing 5.39: Uhr.java

Aufgabe 1 Gegeben sei die folgende Schnittstelle:

```
1  package de.hsrm.cs.oose13;
2
3  import java.awt.Graphics;
4
5  public interface CollisionScene {
6    void collisions();
7    void move();
8    void paintAll(Graphics g);
9
10   int getWidth();
11   int getHeight();
12 }
```

Listing 5.40: CollisionScene.java

Schreiben Sie eine Klasse `GeometricObjectsScene`, die dieses Schnittstelle implementiert. Ein Objekt der Klasse `GeometricObjectsScene` soll eine Reihung (Array) von `GeometricObject`-Objekten im Konstruktor übergeben bekommen. Die Methoden sollen folgende Funktionalität haben:

- in `move` soll jedes Objekt in der Reihung bewegt werden.
- in `paintAll` soll jedes Objekt der Reihung auf das `Graphics`-Objekt gezeichnet werden.
- in `collisions` sollen zunächst alle Objekte der Reihung getestet werden, ob sie außerhalb des Bereichs von 0 bis zu `getWidth()` liegen. Wenn ja, soll die Bewegung in x-Richtung für das Objekt umgedreht werden. Dasselbe soll für die y-Richtung und `getHeight()` gemacht werden.

  Desweiteren sollen paarweise alle Objekte der Reihung getestet werden, ob sie sich berühren. Wenn sich zwei Objekte berühren, sollen ihre Bewegung in x- und y-Richtung umgedreht werden.

Aufgabe 2 Schreiben Sie eine Klasse `ScenePanel`, die eine Unterklasse von `JPanel` ist. Die Klasse soll ein Objekt der Schnittstelle `CollisionScene` im Konstruktor übergeben bekommen.

Überschreiben Sie die folgenden Methoden von `JPanel`:

- `public Dimension getPreferredSize()` : es soll die Dimension aus Weite und Höhe des `CollisionScene`-Objekts zurück gegeben werden.
- `protected void paintComponent(Graphics g)`: es soll erst `paintComponent`der Oberklasse und dann für das `CollisionScene`-Objekt die Methode `paintAll` aufgerufen werden.

Die Klasse soll ein Feld vom Typ `javax.swing.Timer` haben. Dieses soll mit einem `Timer`-Objekt initialisiert werden, das 36 mal in der Sekunde auslöst. Die dabei ausgeführte Aktion soll für das `CollisionScene`-Objekt nacheinander die Methoden `move` und `collisions` und schließlich für den `JPanel` die Methode `repaint` aufrufen.

Erstellen Sie ein Objekt von der Klasse, starten Sie das `Timer`-Objekt und lasse Sie es in einem `JFrame` anzeigen.

Aufgabe 3 Schreiben Sie schließlich eine weitere Unterklasse von `JPanel`. Die Klasse soll `StartStopScenePanel` heißen. Auch hier wird im Konstruktor ein `CollisionScene`-Objekt übergeben. Die Klasse soll zwei GUI-Komponenten enthalten:

- ein Knopfobjekt der Klasse `JButton`.
- ein Objekt Ihrer Klasse `ScenePanel`

Durch Knopfdruck soll der Timer des `ScenePanel`-Objekts gestartet oder wieder angehalten werden.

Erstellen Sie ein Objekt von der Klasse und lassen Sie es in einem `JFrame` anzeigen.

5.5.1 Animationen

Mit dem Prinzip des Timers können wir jetzt auf einfache Weise Animationen realisieren. In einer Animation bewegt sich etwas. Dieses drücken wir durch eine entsprechende Schnittstelle aus:

```
1 package name.panitz.animation;
2
3 public interface Animation {
4   public void move();
5 }
```

Listing 5.41: Animation.java

Wir wollen einen besonderen `JPanel` realisieren, in dem sich etwas bewegen kann. Damit soll ein solcher `JPanel` auch eine Animation sein. Es bietet sich an, eine abstrakte Klasse zu schreiben, in der die Methode `move` noch nicht implementiert ist:

```
1 package name.panitz.animation;
2
3 import javax.swing.JPanel;
4 import javax.swing.Timer;
5 import java.awt.event.*;
6
7 public abstract class AnimatedJPanel
8   extends JPanel implements Animation {
```

Listing 5.42: AnimatedJPanel.java

Um zeitgesteuert das Szenario der Animation zu verändern, brauchen wir einen `Timer`.

```
9   Timer t;
```

Listing 5.43: AnimatedJPanel.java

Im Konstruktor wird dieser initialisiert. Als Ereignisbehandlung wird ein Ereignisbehandlungsobjekt erzeugt, das die Methode `move` aufruft, also dafür sorgt, dass die Szenerie sich weiterbewegt und das dafür sorgt, dass die Szenerie neu gezeichnet wird. Wir starten diesen Timer gleich.

```
10  public AnimatedJPanel(){
11    super(true);
12    t = new Timer(29,new ActionListener(){
13      public void actionPerformed(ActionEvent _){
14        move();
15        repaint();
```

```
16        }
17      });
18      t.start();
19    }
20  }
```

Listing 5.44: AnimatedJPanel.java

Jetzt können wir durch implementieren der Methode `move` und Überschreiben der Methode `paintComponent` beliebige Animationen erzeugen. Als erstes schreiben wir eine Klasse in der ein Kreis sich auf und ab bewegt:

```
1  package name.panitz.animation;
2
3  import java.awt.Graphics;
4  import java.awt.Dimension;
5  import java.awt.Color;
6  import javax.swing.JFrame;
7
8  public class BouncingBall extends AnimatedJPanel {
```

Listing 5.45: BouncingBall.java

Die Größe des Kreises und des Spielfeldes setzen wir in Konstanten fest:

```
9    final int width = 100;
10   final int height = 200;
11   final int ballSize = 20;
```

Listing 5.46: BouncingBall.java

Der Ball soll sich entlang der y-Achse bewegen, und zwar pro Bild um 4 Pixel:

```
12   int yDir = 4;
```

Listing 5.47: BouncingBall.java

Anfangs soll der Ball auf der Hälfte der x-Achse liegen und ganz oben im Bild liegen:

```
13   int ballX = width/2-ballSize/2;
14   int ballY = 0;
```

Listing 5.48: BouncingBall.java

Wir bewegen den Ball. Wenn er oben oder unten am Spielfeldrand anstößt, so ändert er seine Richtung:

```
15   public void move(){
16     if (ballY>height-ballSize || ballY<0) yDir=-yDir;
17     ballY=ballY+yDir;
18   }
```

Listing 5.49: BouncingBall.java

Zum Zeichnen, wird ein roter Hintergrund gezeichnet und der Kreis an seiner aktuellen Position.

```
19   public void paintComponent(Graphics g){
20     g.setColor(Color.RED);
21     g.fillRect(0,0,width,height);
22     g.setColor(Color.YELLOW);
23     g.fillOval(ballX,ballY,ballSize,ballSize);
24   }
```

Listing 5.50: BouncingBall.java

Unsere Größe wird verwendet als bevorzugte Größe der Komponente:

```
25   public Dimension getPreferredSize(){
26     return new Dimension(width,height);
27   }
```

Listing 5.51: BouncingBall.java

Und schließlich folgt eine kleine Hauptmethode zum Starten der Animation.

```
28   public static void main(String [] _){
29     JFrame f = new JFrame("");
30     f.add(new BouncingBall());
31     f.pack();
32     f.setVisible(true);
33   }
34 }
```

Listing 5.52: BouncingBall.java

Aufgabe 1 Schreiben Sie eine Klasse Labyrinth. Ein Labyrinth sei eine Unterklasse von `JPanel` Ein Labyrinth enthalte einen Array von geometrischen Objekten. Diese geometrischen Objekte sind Quadrate. Die Kantenlänge der Quadrate ist in einem Feld der Klasse Labyrinth festgelegt. Das Labyrinth soll die Größe so gesetzt haben, dass für eine bestimmte Anzahl von Zeilen und Spalten der Quadrate Platz ist.

Beispiel: sei die Kantenlänge der Quadrate 20. Das Labyrinth habe 30 Zeilen und 50 Spalten. Dann ist das Labyrinth 50*20=1000 Pixel in x-Richtung und 30*20=600 Pixel in y-Richtung.

Die Klasse `Labyrinth` habe einen Konstruktor, in dem ein Dateiname einer Textdatei übergeben wird. In der Textdatei werden mehrere Zeilen gleicher Länge angenommen. Die Textdatei entspricht der Zeilen und Spalten des Labyrinths. Wenn in der Textdatei an einer Position ein großes 'W' steht, dann ist an dieser Position ein Quadrat zu setzen, ansonsten nicht.

Beispiel.

```
WWWWWWWWWW
W    WW  W
W WWWW   W
W W    W W
W WWWW W W
W        W
WWWWWWWWWW
```

Hier eine kleines Labyrinth mit 10 Spalten und 7 Zeilen und 42 Quadraten für die Wände.

Folgende Klasse enthält eine statische Methode um aus einer Textdatei die einzelnen Zeilen in einen Array zu sammeln.

Schreiben Sie eine Hauptmethode, in der Labyrinthe aus Textdateien eingelesen und in einem `JFrame`-Objekt angezeigt werden.

```
1  package de.hsrm.cs.oose13.util;
2
3  import java.io.*;
4  import java.util.*;
5
6  public class FileUtil {
7
8    public static String[] readTextLines(String fileName){
9      try {
10       return readTextLines(new FileReader(fileName));
11     } catch (IOException e) {
12       e.printStackTrace();
13       throw new RuntimeException(e);
14     }
15   }
16
17   public static String[] readTextLines(Reader fileReader) throws
       IOException {
18     List<String> result = new LinkedList<>();
19     BufferedReader in = new BufferedReader(fileReader);
20     String line=null;
21     while ((line = in.readLine() )!=null){
22       result.add(line);
23     }
24     return result.toArray(new String[0]);
25   }
26   public static void main(String[] args) {
```

```
27      for (String l:readTextLines("test.txt")){
28        System.out.println(l);
29      }
30    }
31  }
```

Listing 5.53: FileUtil.java

Aufgabe 1 Projektaufgabe (20 Punkte)

Sie sollen in dieser abschließenden Projektaufgabe eine kleine Spielapplikation entwickeln. Die Applikation soll mit einem `java.swing.Timer` animiert sein. Es soll auf den Klassen, die im Laufe des Semesters erstellt wurden, aufgebaut werden.

Das Spiel soll die grundlegende Spielidee von PacMan verfolgen. Als Beispiel können Sie PacMan auf folgender Webseite spielen: Google Logo als Pacman (`www.google.com/pacman`) .

Mindestanforderung:

- das Spielfeld ist ein Labyrinth, das aus einer Datei eingelesen wird.
- es gibt eine Spielfigur, die mit den Pfeiltasten gesteuert wird.
- die Spielfigur kann nicht durch Wände laufen.
- es gibt mehrere weitere Figuren (Geister), die sich selbstständig durch das Labyrinth bewegen.
- auch die Geister können normaler Weise nicht durch Wände laufen.
- Berührungen von Geist und Spielfigur führen dazu, dass die Spielfigur stirbt.
- es gibt weitere Objekte, die die Spielfigur einsammeln muss, um Punkte zu bekommen.
- es gibt eine Anzeige, wieviel Punkte gesammelt wurden und wieviel Leben die Spielfigur noch hat.

Ausbauanforderungen:

- Die Spielfigur kann Kraftpillen aufsammeln, mit der sie immun gegen Geister ist und diese zerstören kann, anstatt von ihnen zerstört zu werden.
- alle GeometricObject-Objekte werden durch Bilddateien dargestellt.
- es gibt mehrere Level, die sich in Form des Labyrinth unterscheiden.

Profiaubaustufe:

- es gibt ein Menu, in dem Einstellungen gesetzt werden können.
- es gibt Sound während des Spiels.

Abgabe:

- das Spiel ist in der letzten Praktikumsstunde (letzte Januarwoche) in einem 5 minütigen Vortrag vorzustellen. (unbedingt Länge einhalten und dafür sorgen, dass es keine Verzögerung gibt, das eigene Spiel am Beamer zu zeigen. Wir wollen und können die 90 Minuten der Praktikumsstunde nicht überziehen.)
- letzte Abgabemöglichkeit ist am Sonntag 2. Februar 20:15Uhr per Mail an den Lehrbeauftragten der Praktikumsstunde
- die Abgabe beinhaltet:
  - einen eclipse Projektordner mit alles Resourcen.
  - eine kleine PDF-Datei, die die Besonderheiten der eigenen Lösung zeigen. In diesem Dokument sind alle zusätzlichen Quellen und Hilfen, die benutzt worden, genannt. Insbesondere die Urheber der Bilddateien, falls welche verwendet wurden. Es ist darin zu unterschreiben, dass die Aufgabe eigenständig gelöst wurde und keine anderen als die angegebenen Hilfen verwendet wurden.
  - der Eclispe-Projektordner und das PDF-Dokument sind in einer tar-Datei zu verpacken. Die tar-Datei hat den Namen, der aus Ihren Nachnamen gefolgt von Ihrem Vornamen besteht. Verwenden sie keine Umlaute oder sonstigen Sonderzeichen in dem dateinamen, also z.B. `muellerMartin.tar`.

5.6 Weitere Swing Komponenten

Um einen kleinen Überblick der vorhandenen Swing Komponenten zu bekommen, können wir ein kleines Programm schreiben, daß möglichst viele Komponenten einmal instanziiert. Hierzu schreiben wir eine kleine Testklasse:

```
1 package name.panitz.gui.example;
2
3 import javax.swing.*;
4 import java.awt.*;
5 import java.util.*;
6
7 public class ComponentOverview  {
8   public ComponentOverview(){
```

Listing 5.54: ComponentOverview.java

Darin definieren wir eine Reihung von einfachen Swing-Komponenten:

```
9     JComponent [] cs1 =
10     {new JButton("knopf")
11     ,new JCheckBox("check mich")
```

```
12      ,new JRadioButton("drück mich")
13      ,new JMenuItem("ins Menue mit mir")
14      ,new JComboBox(combos)
15      ,new JList(combos)
16      ,new JSlider(0,350,79)
17      ,new JSpinner(new SpinnerNumberModel(18,0.0,42.0,2.0))
18      ,new JTextField(12)
19      ,new JFormattedTextField("hallo")
20      ,new JLabel("einfach nur ein Label")
21      ,new JProgressBar(0,42)
22      };
```

Listing 5.55: ComponentOverview.java

Sowie eine zweite Reihung von komplexeren Swing-Komponenten:

```
23    JComponent [] cs2 =
24     {new JColorChooser(Color.RED)
25     ,new JFileChooser()
26     ,new JTable(13,5)
27     ,new JTree()
28     };
```

Listing 5.56: ComponentOverview.java

Diese beiden Reihungen zeigen sollen mit einer Hilfsmethode in einem Fenster angezeigt werden:

```
29    displayComponents(cs1,3);
30    displayComponents(cs2,2);
31   }
```

Listing 5.57: ComponentOverview.java

Für die Listen- und Auswahlkomponenten oben haben wir eine Reihung von Strings benutzt:

```
32   String [] combos = {"friends","romans","contrymen"};
```

Listing 5.58: ComponentOverview.java

Bleibt die Methode zu schreiben, die die Reihungen von Komponenten anzeigen kann. Als zweites Argument bekommt diese Methode übergeben, in wieviel Spalten die Komponenten angezeigt werden sollen.

```
33   public void displayComponents(JComponent [] cs,int col){
```

Listing 5.59: ComponentOverview.java

Ein Fenster wird definiert, für das eine Gridlayout-Zwischenkomponente mit genügend Zeilen erzeugt wird:

```
34      JFrame f = new JFrame();
35      JPanel panel = new JPanel();
36      panel.setLayout(
37       new GridLayout(cs.length/col+(cs.length%col==0?0:1),col));
```

Listing 5.60: ComponentOverview.java

Für jede Komponente wird ein `Panel` mit Rahmen und den Klassennamen der Komponente als Titel erzeugt und der Zwischenkomponente hinzugefügt:

```
38      for (JComponent c:cs){
39        JPanel p = new JPanel();
40        p.add(c);
41        p.setBorder(BorderFactory
42                    .createTitledBorder(c.getClass().getName()));
43        panel.add(p);
44      }
```

Listing 5.61: ComponentOverview.java

Schließlich wird noch das Hauptfester zusammengepackt:

```
45      f.getContentPane().add(panel);
46      f.pack();
47      f.setVisible(true);
48    }
```

Listing 5.62: ComponentOverview.java

Und um alles zu starten, noch eine kleine Hauptmethode:

```
49    public static void main(String [] _){
50      new  ComponentOverview();
51    }
52  }
```

Listing 5.63: ComponentOverview.java

Wir erhalten einmal die Übersicht von Komponenten wie in Abbildung 5.5 und einmal wie in Abbildung 5.6 dargestellt.

Womit wir uns noch nicht im einzelnen beschäftigt haben, ist das Datenmodell, das hinter den einzelnen komplexeren Komponenten steht.

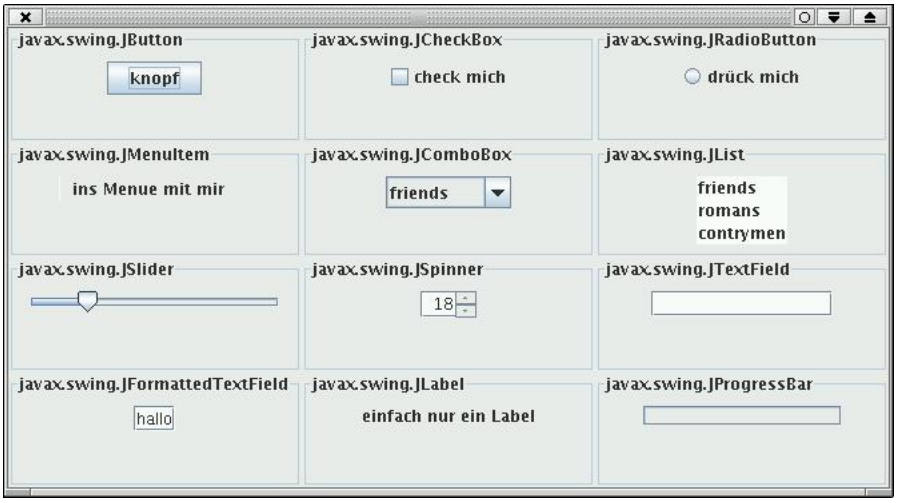

Abbildung 5.5: Überblick über einfache Komponenten.

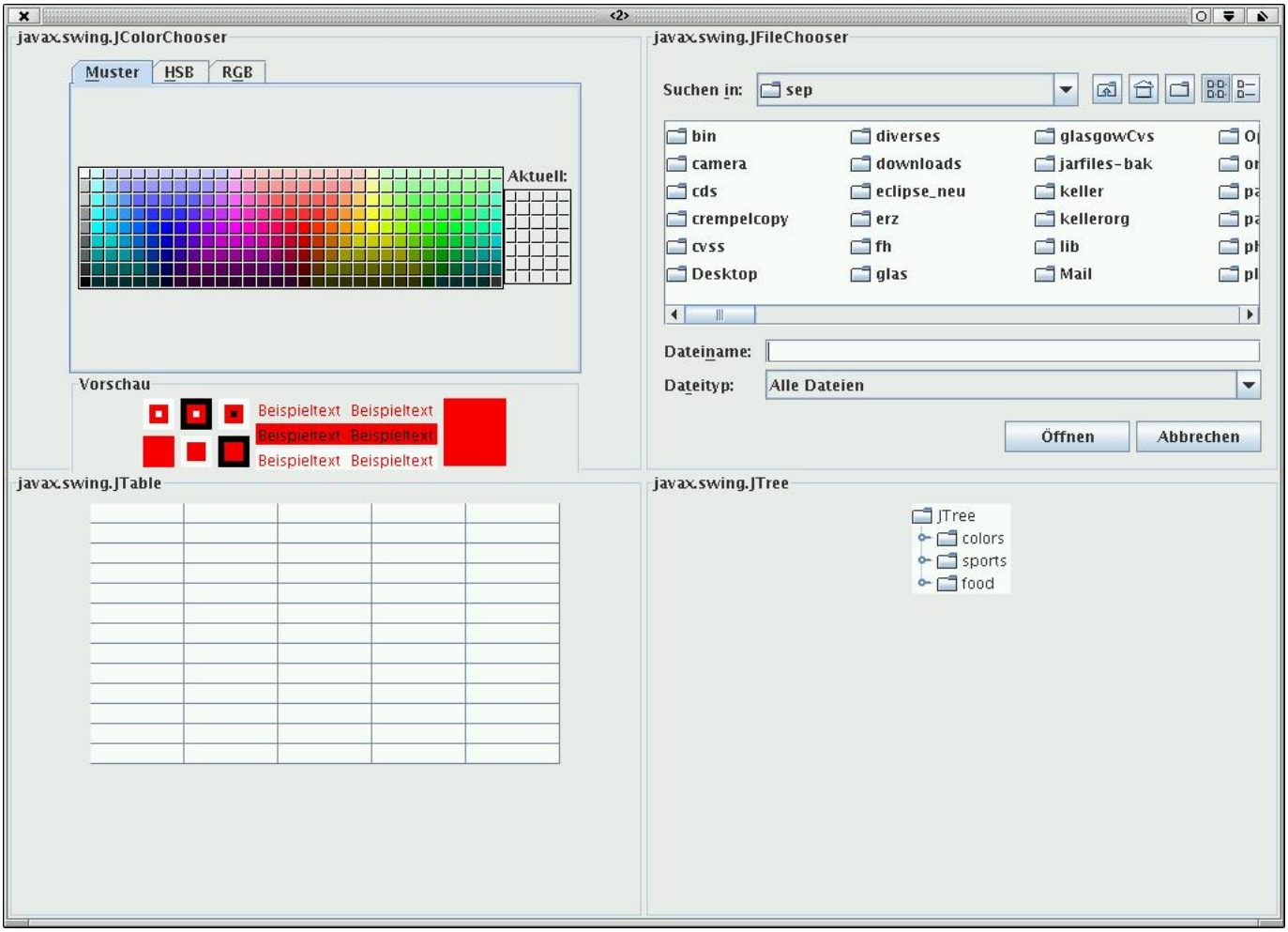

Abbildung 5.6: Überblick über komplexere Komponenten.

Kapitel 6

Weiterführende Konzepte

6.1 Reihungen (Arrays)

Java kennt, wie fast alle Programmiersprachen, ein weiteres Konzept von Sammlungen: Reihungen (eng. arrays).[1] Reihungen stellen im Gegensatz zu Listen oder Mengen eine Menge von Daten gleichen Typs mit fester Anzahl dar. Jedes Element einer Reihung hat einen festen Index, über den es direkt angesprochen werden kann.

6.1.1 Deklaration von Reihungen

Eine Reihung hat einen festen Elementtyp.

Ein Reihungstyps wird deklariert, indem dem Elementtyp ein eckiges Klammernpaar nachgestellt wird, z.B. ist String [] eine Reihung von Stringelementen. Die Elemente einer Reihung können sowohl von einem Objekttyp als auch von einem primitiven Typ sein, also gibt es auch den Typ int [] oder z.B. boolean [].

Reihungen sind Objekte. Sie sind zuweisungskompatibel für Objektfelder, es lassen sich Typzusicherungen auf Reihungen durchführen und Reihungen haben ein Feld `lengthvom` Typ `int`, das die feste Länge einer Reihung angibt.

```
1 class ObjectArray {
2   public static void main(String [] args){
3     Object as = args;
4     System.out.println(((String [])as).length);
5   }
6 }
```

Listing 6.1: ObjectArray.java

[1] In der deutschen Literatur findet man oft den Ausdruck *Datenfeld* für Reihungen. Wir haben uns gegen diesen Ausdruck entschieden, um nicht mit Feldern einer Klasse durcheinander zu kommen.

6.1.2 Erzeugen von Reihungen

Es gibt zwei Verfahren, um Reihungen zu erzeugen: indem die Elemente der Reihung aufgezählt werden oder indem die Länge der Reihung angegeben wird. Eine Mischform, in der sowohl Länge als auch die einzelnen Elemente angegeben werden, gibt es nicht.

Die einfachste Art, um eine Reihung zu erzeugen, ist, die Elemente aufzuzählen. Hierzu sind die Elemente in geschweiften Klammern mit Komma getrennt aufzuzählen:

```
1  class FirstArray {
2
3   static String [] komponisten
4    = {"carcassi","carulli","giuliani"
5       ,"molino","monzino","paganini","sor"};
6
7    public static void main(String [] args){
8      System.out.println(komponisten.length);
9      System.out.println(komponisten.toString());
10   }
11 }
```

Listing 6.2: FirstArray.java

Wie man beim Starten dieser kleinen Klasse erkennen kann, ist für Reihungen keine eigene Methode `toString` in Java implementiert worden.

Eine weitere Methode zur Erzeugung von Reihungen ist, noch nicht die einzelnen Elemente der Reihung anzugeben, sondern nur die Anzahl der Elemente:

```
1  class SecondArray {
2
3    static int [] zahlenReihung = new int[10];
4
5    public static void main(String [] args){
6      System.out.println(zahlenReihung);
7    }
8  }
```

Listing 6.3: SecondArray.java

6.1.3 Zugriff auf Elemente

Die einzelnen Elemente einer Reihung können über einen Index angesprochen werden. Das erste Element einer Reihung hat den Index `0`, das letzte Element den Index `length-1`.

Als Syntax benutzt Java die auch aus anderen Programmiersprachen

bekannte Schreibweise mit eckigen Klammern:

```
String [] stra = {"hallo","welt"};
String str = stra[1];
```

Typischer Weise wird mit einer `for`-Schleife über den Index einer Reihung iteriert. So läßt sich z.B. eine Methode,

die eine Stringdarstellung für Reihungen erzeugt, wie folgt schreiben:

```
public class ArrayToString{
  static public String arrayToString(Object[] obja){
    StringBuffer result = new StringBuffer("{");

    for (int i=0;i<obja.length;i=i+1){
      if (i>0) result.append(",");
      result.append(obja[i].toString());
    }

    result.append("}");
    return result.toString();
  }
}
```

Listing 6.4: ArrayToString.java

6.1.4 Ändern von Elementen

Eine Reihung kann als ein Komplex von vielen einzelnen Feldern gesehen werden. Die Felder haben keine eigenen Namen, sondern werden

über den Namen der Reihung zusammen mit ihrem Index angesprochen. Mit diesem Bild ergibt sich automatisch, wie nun einzelnen Reihungselementen neue Objekte zugewiesen werden können:

```
String [] stra = {"hello","world"};
stra[0]="hallo";
stra[1]="welt";
```

6.1.5 Die For-Each Schleife

Eine häufige Aufgabe ist, für alle Elemente einer Reihung eine bestimmte Aktion durchzuführen. Wir haben schon die typische for-Schleife über den Index der Elemente gesehen. Java bietet aber für den Zweck, um über alle Elemente eines Array

(oder sonstigen Sammlung) zu iterieren eine spezielle Variante der for-Schleife an, die sogenannte for-each Schleife.

```
1  package name.panitz.oose;
2  public class ForEachArray{
3    public static void main(String[] args){
4      int[] xs = {17, 4, 42, 18, -6};
5
6      for (int x:xs){
7        System.out.println(x*x);
8      }
9    }
10 }
```

Listing 6.5: ForEachArray.java

6.2 Generische Typen

Generische Typen wurden im JSR014 definiert. In der Expertengruppe des JSR014 war der Autor dieses Skripts zeitweilig als Stellvertreter der Software AG Mitglied. Die Software AG hatte mit der Programmiersprache Bolero bereits einen Compiler für generische Typen implementiert. Der Bolero Compiler generiert auch Java Byte Code. Von dem ersten Wunsch nach Generizität bis zur nun vorliegenden Javaversion 1.5 sind viele Jahre vergangen. Andere wichtige JSRs, die in Java 1.5 integriert werden, tragen bereits die Nummern 175 und 201. Hieran kann man schon erkennen, wie lange es gedauert hat, bis generische Typen in Java integriert wurden.

Interessierten Programmierern steht schon seit Mitte der 90er Jahre eine Javaerweiterung mit generischen Typen zur Verfügung. Unter den Namen Pizza existiert eine Javaerweiterung, die nicht nur generische Typen, sondern auch algebraische Datentypen mit *pattern matching* und Funktionsobjekten zu Java hinzufügte. Unter den Namen *GJ* für *Generic Java* wurde eine allein auf generische Typen abgespeckte Version von *Pizza* publiziert. *GJ* ist tatsächlich der direkte Prototyp für Javas generische Typen. Die Expertenrunde des JSR014 hat *GJ* als Grundlage für die Spezifikation genommen und an den grundlegenden Prinzipien auch nichts mehr geändert.

6.2.1 Generische Klassen

Die Idee für generische Typen ist, eine Klasse zu schreiben, die für verschiedene Typen als Inhalt zu benutzen ist. Das geht bisher in Java, allerdings mit einem kleinen Nachteil. Versuchen wir einmal, in traditionellem Java eine Klasse zu schreiben, in der wir beliebige Objekte speichern können. Um beliebige Objekte

speichern zu können, brauchen wir ein Feld, in dem Objekte jeden Typs gespeichert werden können. Dieses Feld muss daher den Typ `Object` erhalten:

```
class OldBox {
  Object contents;
  OldBox(Object contents){this.contents=contents;}
}
```

Listing 6.6: OldBox.java

Der Typ `Object` ist ein sehr unschöner Typ; denn mit ihm verlieren wir jegliche statische Typinformation. Wenn wir die Objekte der Klasse `OldBox` benutzen wollen, so verlieren wir sämtliche Typinformation über das in dieser Klasse abgespeicherte Objekt. Wenn wir auf das Feld `contents` zugreifen, so haben wir über das darin gespeicherte Objekte keine spezifische Information mehr. Um das Objekt weiter sinnvoll nutzen zu können, ist eine dynamische Typzusicherung durchzuführen:

```
class UseOldBox{
  public static void main(String [] _){
    OldBox b = new OldBox("hello");
    String s = (String)b.contents;
    System.out.println(s.toUpperCase());
    System.out.println(((String) s).toUpperCase());
  }
}
```

Listing 6.7: UseOldBox.java

Wann immer wir mit dem Inhalt des Felds `contents` arbeiten wollen, ist die Typzusicherung während der Laufzeit durchzuführen. Die dynamische Typzusicherung kann zu einem Laufzeitfehler führen. So übersetzt das folgende Programm fehlerfrei, ergibt aber einen Laufzeitfehler:

```
class UseOldBoxError{
  public static void main(String [] _){
    OldBox b = new OldBox(new Integer(42));
    String s = (String)b.contents;
    System.out.println(s.toUpperCase());
  }
}
```

Listing 6.8: UseOldBoxError.java

```
sep@linux:~/fh/java1.5/examples/src> javac UseOldBoxError.java
sep@linux:~/fh/java1.5/examples/src> java UseOldBoxError
Exception in thread "main" java.lang.ClassCastException
        at UseOldBoxError.main(UseOldBoxError.java:4)
sep@linux:~/fh/java1.5/examples/src>
```

Wie man sieht, verlieren wir Typsicherheit, sobald der Typ `Object` benutzt wird. Bestimmte Typfehler können nicht mehr statisch zur Übersetzungszeit, sondern erst dynamisch zur Laufzeit entdeckt werden.

Der Wunsch ist, Klassen zu schreiben, die genauso allgemein benutzbar sind wie die Klasse `OldBox` oben, aber trotzdem die statische Typsicherheit garantieren, indem sie nicht mit dem allgemeinen Typ `Object` arbeiten. Genau dieses leisten generische Klassen. Hierzu ersetzen wir in der obigen Klasse jedes Auftreten des Typs `Object` durch einen Variablennamen. Diese Variable ist eine Typvariable. Sie steht für einen beliebigen Typen. Dem Klassennamen fügen wir zusätzlich in der Klassendefinition in spitzen Klammern eingeschlossen hinzu, daß diese Klasse eine Typvariable benutzt. Wir erhalten somit aus der obigen Klasse `OldBox` folgende generische Klasse `Box`.

```
1 class Box<ET> {
2   ET contents;
3   Box(ET  contents){this.contents=contents;}
4 }
```

Listing 6.9: Box.java

Die Typvariable `ET` ist als allquantifiziert zu verstehen. Für jeden Typ `ET` können wir die Klasse `Box` benutzen. Man kann sich unsere Klasse `Box` analog zu einer realen Schachtel vorstellen: Beliebige Dinge können in die Schachtel gelegt werden. Betrachten wir dann allein die Schachtel von außen, können wir nicht mehr wissen, was für ein Objekt darin enthalten ist. Wenn wir viele Dinge in Schachteln packen, dann schreiben wir auf die Schachtel jeweils drauf, was in der entsprechenden Schachtel enthalten ist. Ansonsten würden wir schnell die Übersicht verlieren. Und genau das ermöglichen generische Klassen. Sobald wir ein konkretes Objekt der Klasse `Box` erzeugen wollen, müssen wir entscheiden, für welchen Inhalt wir eine Box brauchen. Dieses geschieht, indem in spitzen Klammern dem Klassennamen `Box` ein entsprechender Typ für den Inhalt angehängt wird. Wir erhalten dann z.B. den Typ `Box<String>`, um Strings in der Schachtel zu speichern, oder `Box<Integer>`, um Integerobjekte darin zu speichern:

```
1  class UseBox{
2    public static void main(String [] _){
3      Box<String> b1 = new Box<String>("hello");
4      String s = b1.contents;
5      System.out.println(s.toUpperCase());
6      System.out.println(b1.contents.toUpperCase());
7
8      Box<Integer> b2 = new Box<Integer>(new Integer(42));
9
10     System.out.println(b2.contents.intValue());
11   }
12 }
```

Listing 6.10: UseBox.java

Wie man im obigen Beispiel sieht, fallen jetzt die dynamischen Typzusicherungen weg. Die Variablen `b1` und `b2` sind jetzt nicht einfach vom Typ `Box`, sondern vom Typ `Box<String>` respektive `Box<Integer>`.

Da wir mit generischen Typen keine Typzusicherungen mehr vorzunehmen brauchen, bekommen wir auch keine dynamischen Typfehler mehr. Der Laufzeitfehler, wie wir ihn ohne die generische Box hatten, wird jetzt bereits zur Übersetzungszeit entdeckt. Hierzu betrachte man das analoge Programm:

```
1 class UseBoxError{
2   public static void main(String [] _){
3     Box<String> b = new Box<String>(new Integer(42));
4     String s = b.contents;
5     System.out.println(s.toUpperCase());
6   }
7 }
```

Die Übersetzung dieses Programms führt jetzt bereits zu einen statischen Typfehler:

```
sep@linux:~/fh/java1.5/examples/src> javac UseBoxError.java
UseBoxError.java:3: cannot find symbol
symbol  : constructor Box(java.lang.Integer)
location: class Box<java.lang.String>
    Box<String> b = new Box<String>(new Integer(42));
                        ^
1 error
sep@linux:~/fh/java1.5/examples/src>
```

6.2.2 Vererbung

Generische Typen sind ein Konzept, das orthogonal zur Objektorientierung ist. Von generischen Klassen lassen sich in gewohnter Weise Unterklassen definieren. Diese Unterklassen können, aber müssen nicht selbst generische Klassen sein. So können wir unsere einfache Schachtelklasse erweitern, so dass wir zwei Objekte speichern können:

```
1 class GPair<A,B> extends Box<A>{
2   GPair(A x,B y){
3     super(x);
4     snd = y;
5   }
6
```

```
7   B snd;
8
9   public String toString(){
10    return "("+contents+","+snd+")";
11  }
12 }
```

Listing 6.11: GPair.java

Die Klasse `GPair` hat zwei Typvariablen. Instanzen von `GPair` müssen angeben von welchem Typ die beiden zu speichernden Objekte sein sollen.

```
1  class UsePair{
2    public static void main(String [] _){
3      GPair<String,Integer> p
4       = new GPair<String,Integer>("hallo",new  Integer(40));
5
6      System.out.println(p);
7      System.out.println(p.contents.toUpperCase());
8      System.out.println(p.snd.intValue()+2);
9    }
10 }
```

Listing 6.12: UsePair.java

Wie man sieht kommen wir wieder ohne Typzusicherung aus. Es gibt keinen dynamischen Typcheck, der im Zweifelsfall zu einer Ausnahme führen könnte.

```
sep@linux:~/fh/java1.5/examples/classes> java UsePair
(hallo,40)
HALLO
42
sep@linux:~/fh/java1.5/examples/classes>
```

Wir können auch eine Unterklasse bilden, indem wir mehrere Typvariablen zusammenfassen. Wenn wir uniforme Paare haben wollen, die zwei Objekte gleichen Typs speichern, können wir hierfür eine spezielle Paarklasse definieren.

```
1  class UniPair<A> extends GPair<A,A>{
2    UniPair(A x,A y){super(x,y);}
3    void swap(){
4      final A z = snd;
5      snd = contents;
6      contents = z;
7    }
8  }
```

Listing 6.13: UniPair.java

Da beide gespeicherten Objekte jeweils vom gleichen Typ sind, konnten wir jetzt eine Methode schreiben, in der diese beiden Objekte ihren Platz tauschen. Wie man sieht, sind Typvariablen ebenso wie unsere bisherigen Typen zu benutzen. Sie können als Typ für lokale Variablen oder Parameter genutzt werden.

```
1  class UseUniPair{
2    public static void main(String [] _){
3      UniPair<String> p
4       = new UniPair<String>("welt","hallo");
5
6      System.out.println(p);
7      p.swap();
8      System.out.println(p);
9    }
10 }
```

Listing 6.14: UseUniPair.java

Wie man bei der Benutzung der uniformen Paare sieht, gibt man jetzt natürlich nur noch einen konkreten Typ für die Typvariablen an. Die Klasse `UniPair` hat ja nur eine Typvariable.

```
sep@linux:~/fh/java1.5/examples/classes> java UseUniPair
(welt,hallo)
(hallo,welt)
sep@linux:~/fh/java1.5/examples/classes>
```

Wir können aber auch Unterklassen einer generischen Klasse bilden, die nicht mehr generisch ist. Dann leiten wir für eine ganz spezifische Instanz der Oberklasse ab. So läßt sich z.B. die Klasse `Box` zu einer Klasse erweitern, in der nur noch Stringobjekte verpackt werden können:

```
1  class StringBox extends Box<String>{
2    StringBox(String x){super(x);}
3  }
```

Listing 6.15: StringBox.java

Diese Klasse kann nun vollkommen ohne spitze Klammern benutzt werden:

```
1  class UseStringBox{
2    public static void main(String [] _){
3      StringBox b = new StringBox("hallo");
4      System.out.println(b.contents.length());
5    }
6  }
```

Listing 6.16: UseStringBox.java

6.2.3 Beispiel einer eigenen Listenklasse

Die Paradeanwendung für generische Typen sind sogenannte Container- oder Sammlungsklassen, d.h. Klassen wie die erste generische Klasse `Box` die als Behälter für ein oder mehrere Elemente eines variabel gehaltenen Typs dienen. Die Klassen `Box` und `Pair` sind in dieser Hinsicht recht langweilig, da sie nur ein bzw. zwei Elemente beinhalten. Eine häufig benötigte Funktionalität ist die Sammlung von potentiell beliebig vielen Elementen in einer bestimmten Reihenfolge. Eine solche Sammlung von Elementen wird als Liste bezeichnet. Mit Hilfe der generischen Typen können wir eine eigene generische Listenklasse implementieren.

Der Trick dabei ist, sich der Reihungen als internen Speicher zu bedienen. Reihungen sind ja in der Lage sind Elemente in einer festen Reihenfolge abzuspeichern. Das Problem der Reihungen ist aber, dass sie im Vorfeld beim Erzeugen nur für eine maximale Anzahl von Elementen erzeugt werden können. Eine Reihung kann dynamisch nicht nachträglich vergrößert werden.

In der Implementierung einer Listenklasse mit Hilfe einer Reihung geht man so vor, dass es eine interne Reihung gibt. Ist diese Reihung zu klein geworden, wird eine neue, größere Reihung erzeugt und alle Elemente in diese umkopiert.

Unsere Listenklasse benötigt also zwei interne Felder:

- eines, das die Reihung als Datenspeicher enthält
- eines, das notiert, wie viele Elemente in der Liste bereits enthalten sind.

```
1 package name.panitz.util;
2
3 public class OurList<A>{
4   private Object[] store =  new Object[10];
5
6   private int theSize = 0;
```

Listing 6.17: OurList.java

Die wichtigste Methode, die wir für eine Liste benötigen, soll ein Element in die Liste anfügen. Hierzu schreiben wir die Methode `add`, die ein Element am Ende der Liste hinzufügen soll.

```
7   public boolean add(A el){
```

Listing 6.18: OurList.java

Die Methode hat ein Problem, wenn die Reihung, die als Datenspeicher benutzt wird bereits komplett mit Listenelementen gefüllt

```
8     if (theSize >= store.length){
9       Object[] newStore = new Object[store.length+10];
10
11      for (int i= 0;i<store.length;i++){
12        newStore[i] = store[i];
13      }
14      store = newStore;
15    }
```

Listing 6.19: OurList.java

```
16    store[theSize] = el;
17    theSize++;
18    return true;
19  }
```

Listing 6.20: OurList.java

```
20  public int size() {
21    return theSize;
22  }
```

Listing 6.21: OurList.java

```
23  public A get(int i){
24    return (A)store[i];
25  }
```

Listing 6.22: OurList.java

```
26  public static void main(String[] args) {
27    OurList<String> xs = new OurList<>();
28    xs.add("hallo");
29    xs.add("Freunde");
30    xs.add("Römer");
31    xs.add("Landsleute");
32    xs.add("leiht");
33    xs.add("mir");
34    xs.add("Euer");
35    xs.add("Ohr");
36    xs.add("Landsleute");
37    xs.add("friends");
38    xs.add("Romans");
39    System.out.println(xs.size());
40    System.out.println(xs.get(2));
41
42    for (int i = 0; i < xs.size(); i++) {
43      System.out.println(xs.get(i).toUpperCase());
```

```
44        }
45    }
46 }
```

Listing 6.23: OurList.java

6.2.4 Standard Sammlungsklassen

Java stellt im Paket `java.util` Implementierungen von Sammlungsklassen zur Verfügung. Dabei handelt ich sich um Klassen für Listen, Mengen und Abbildungen. Dieses sind Paradeanwendungen für generische Typen, denn es lassen sich über den Typparameter angeben, welchen Type die Elemente einer Sammlung im speziellen Fall haben.

Die Sammlungsklassen sind über verschiedene Schnittstellen definiert. Die Oberschnittstelle für Sammlungsklassen ist: `java.util.Collection`. Ihre Hauptunterschnittstellen sind: `List` und `Set` für die Darstellung von Listen bzw. Mengen.

Listen

Javas Standardlisten sind durch die Schnittstelle `jva.util.List` definiert. In dieser Schnittstelle finden sich eine Reihe von Methoden.

Zum einen Methoden, um Eigenschaften eines Listenobjekts zu erfragen, wie z.B.:

- `get(int index)` zum Erfragen eines Objektes an einem bestimmten Index der Liste.
- `size()`, um die Länge zu erfragen.
- `contains(Object o)`, um zu testen, ob ein bestimmtes Element in der Liste enthalten ist.

Desweiteren Methoden, die den Inhalt eines Listenobjektes verändern, wie z.B.:

- `add` zum Hinzufügen von Elementen.
- `clear` zum Löschen aller Elemente.
- `remove` zum Löschen einzelner Elemente.

Die eigentlichen konkreten Klassen, die Listen implementieren, sind: `ArrayList`, `LinkedList` und `Vector`. Dabei ist `ArrayList` die gebräuchlichste Implementierung. `Vector` ist eine ältere Implementierung, die als Nachteil hat, dass sie stets eine Synchronisation für nebenläufige Steuerfäden vornimmt, die in der Regel nicht unbedingt benötigt wird.

Die Klasse `ArrayList` entspricht der Klasse `OurList`, die im letzten Abschnitt entwickelt wurde. Es wird also intern eine Reihung benutzt, um die Elemente der Liste abzuspeichern. Wenn diese Reihung zu klein geworden ist, um alle Elemente

der Liste hinzuzufügen, dann wird intern eine größere Reihung angelegt und die Elemente in diesen kopiert.

Die Klasse `LinkedList` hingegen realisiert die Liste auf eine gänzlich andere Art. Hier wird für jedes Element der Liste ein eigenes Listenkettenobjekt erzeugt. Diese Kettenglieder sind dann miteinander verbunden. Es entsteht eine rekursive Struktur. Dieses werden wir im zweiten Semester selbst programmieren.

Warum gibt es zwei verschiedene Listenklassen? Beide realisieren dieselbe Funktionalität. Jedoch haben sie beide sehr unterschiedliches Laufzeitverhalten:

- die `ArrayList` kann effizient auf beliebige Elemente in der Liste zugreifen. Die Methode `get(i)` hat einen konstanten Aufwand. Bei der `LinkedList` hingegen kann man über das Durchlaufen aller Elemente vom ersten Element an, an das i-te Element gelangen. Hier hat die Methode `get(i)` einen linearen Aufwand.
- `LinkedList` kann dafür im Vergleich effizienter Elemente hinzufügen oder löschen. Gerade beim Löschen eines Elements müssen in der `ArrayList` alle nachfolgenden Elemente in der internen Reihung umkopiert werden. Bei einer `LinkedList` kann ein Kettenglied einfach ausgehängt werden

Folgende Klasse zeigt, wie eine Liste erzeugt wird und ihr nach und nach Elemente hinzugefügt werden:

```
1  import java.util.List;
2  import java.util.ArrayList;
3
4  class ListUsage{
5
6    public static void main(String [] args){
7      List<String> xs = new ArrayList<String>();
8         //oder
9        //  = new LinkedList<String>();
10     xs.add("hallo");
11     xs.add("welt");
12     xs.add("wie");
13     xs.add("geht");
14     xs.add("es");
15     xs.add("dir");
16     System.out.println(xs);
17   }
18
19 }
```

Listing 6.24: ListUsage.java

Das Programm hat folgende Ausgabe:

```
sep@swe10:~/fh/beispiele> java ListUsage
[hallo , welt , wie , geht , es , dir ]
sep@swe10:~/fh/beispiele>
```

Wie man sieht, fügt die Methode `add` Objekte am Ende einer Liste an.

Iterieren über Sammlungen

```
1  package name.panitz.util.test;;
2
3  import java.util.ArrayList;
4  import java.util.Collections;
5  import java.util.Iterator;
6  import java.util.LinkedList;
7  import java.util.List;
8
9  public class IterateList {
10   public static void main(String[] args) {
11     List<String> xs = new LinkedList<>();
12     xs.add("hallo");
13     xs.add("Freunde");
14     xs.add("Römer");
15     xs.add("Landsleute");
16     xs.add("leiht");
17     xs.add("mir");
18     xs.add("Euer");
19     xs.add("Ohr");
20     xs.add("Landsleute");
21     xs.add("friends");
22     xs.add("Romans");
23
24
25     System.out.println(xs.size());
26     System.out.println(xs.get(2));
27
28     /* So besser nicht Iterieren, denn get(i) kann teuer sein
29     for (int i = 0; i < xs.size(); i++) {
30       System.out.println(xs.get(i).toUpperCase());
31     }
32     */
33
34     //die Liste Sortieren
35     Collections.sort(xs);
36
37     System.out.println("jetzt mit for Each");
38     //lieber mit for each
39     for (String x : xs) {
40       System.out.println(x);
41     }
42
43     // oder explizit mit dem Iterator
44     System.out.println("jetzt mit Iterator");
45     for (Iterator<String> it = xs.iterator();it.hasNext();  ){
46       String x = it.next();
47       System.out.println(x.toUpperCase());
48     }
49
50     //oder mit forEach Methode und Lambda-Ausdruck
```

```
51      System.out.println("jetzt mit Lambda");
52      xs.forEach(x -> System.out.println(x.toUpperCase()));
53
54      //oder sogar parallelisiert
55      System.out.println("jetzt parallel");
56      xs.parallelStream().forEach(x -> System.out.println(x.toUpperCase
        ()));
57    }
58  }
```

Listing 6.25: IterateList.java

Abbildungen

Streams

6.3 Ein- und Ausgabe

Im Paket `java.io` befinden sich eine Reihe von Klassen, die Ein- und Ausgabe von Daten auf externen Datenquellen erlauben. In den häufigsten Fällen handelt es sich hierbei um Dateien.

Ein-/Ausgabeoperationen werden in Java auf sogenannten Datenströmen ausgeführt. Datenströme verbinden das Programm mit einer externen Datenquelle bzw.Senke. Auf diesen Strömen stehen Methoden zum Senden (Schreiben) von Daten an die Senke bzw. Empfangen (Lesen) von Daten aus der Quelle. Typischer Weise wird ein Datenstrom angelegt, geöffnet, dann werden darauf Daten gelesen und geschrieben und schließlich der Strom wieder geschlossen.

Ein Datenstrom charakterisiert sich dadurch, dass er streng sequentiell arbeitet. Daten werden also auf Strömen immer von vorne nach hinten gelesen und geschrieben.

Java unterscheidet zwei fundamentale unterschiedliche Arten von Datenströmen:

- **Zeichenenströme**: Die Grundeinheit der Datenkommunikation sind Buchstaben und andere Schriftzeichen. Hierbei kann es sich um beliebige Zeichen aus dem Unicode handeln, also Buchstaben so gut wie jeder bekannten Schrift, von lateinischer über kyrillische, griechische, arabische, chinesische bis hin zu exotischen Schriften wie der keltischen Keilschrift. Dabei ist entscheidend, in welcher Codierurung die Buchstaben in der Datenquelle vorliegen. Die Codierung wird in der Regel beim Konstruieren eines Datenstroms festgelegt. Geschieht dieses nicht, so wird die Standardcodierung des Systems, auf dem das Programm läuft, benutzt.
- **Byteströme (Oktettströme)**: Hierbei ist die Grundeinheit immer ein Byte, das als Zahl verstanden wird.

Vier abstrakte Klassen sind die Grundlagen der strombasierten Ein-/Ausgabe in Java. Die Klassen `Reader` und `Writer` zum Lesen bzw. Schreiben von Textdaten und die Klassen `InputStreamReader` und `OutputStreamReader` zum Lesen und Schreiben von Binärdaten. Folgende kleine Tabelle stellt dieses noch einmal dar.

	Eingabe	**Ausgabe**
binär	`InputStream`	`OutputStream`
Texte	`Reader`	`Writer`

Alle vier Klassen haben entsprechend ihrer Funktion Methoden `read` und `write`, die sich auf die enstprechenden Daten beziehen, sprich, ob `byte`-Werte gelesen und geschrieben werden oder `char`-Werte.

Aufgabe 1 Schreiben Sie eine mehrfach überladene statische Methode, die es erlaubt einer Liste aller Dateien, Unterdateien und Ordner sowie wiederum deren Unterdateien und Unterordmer ausgehend von einem Startverzeichnis in einer Liste zu sammeln. Wird kein Startverzeichnis angegeben, so ist das aktuelle Arbeitsverzeichnis als Ausgangspunkt zu verwenden. Die Methoden sollen folgende Signaturen haben.

```
static public List<File> getFileList()

public static List<File> getFileList(String fileName)

public static List<File> getFileList(File startFile)

public static List<File> getFileList(File startFile, List<File> result)
```

Hinweis: lösen sie diese Aufgabe rekursiv. Alle Methode dürfen mit `throws Exception` markiert werden und somit beliebig Ausnahmen werden.

Aufgabe 2 Schreiben Sie folgende Methode:

```
static void convert(String inFile, String outFile, String inEnc,String outEnc)
      throws Exception
```

Es soll eine Textdatei für eine bestimmte Zeichencodierung eingelesen und der gelesene Inhalt in andere andere Textdatei in einer bestimmten Zeichencodierung wieder abgespeichert werden. Die Datei wird also kopiert und dabei eventuell die Zeichencodierung geändert.

6.3.1 Dateibasierte Ein-/Ausgabe

Die vier obigen Grundklassen sind abstrakt, d.h. es können keine Objekte dieser Klassen mit einem Konstruktor erzeugt werden. Wir benötigen Unterklassen dieser

vier Klassen, die die abstrakten Methoden implementieren. Eine der üblichsten Ein-/Ausgabe-Operation bezieht sich auf Dateien. Für alle für der Grundklassen gibt es im API eine Klasse, die die entsprechende Operation für Dateien realisiert. Diese Unterklassen haben im Namen einfach das Wort `File` voran gestellt.

Die einfachste Art der Eingabe dürfte somit das Lesen aus einer Textdatei sein. Hierzu eine einfache Methoden zum Lesen einer Textdatei:

```
1 import java.io.*;
2 package name.panitz.oose.io;
3 public class Textlesen {
```

Listing 6.26: TextLesen

Das Problem mit sämtlichen Ein-/Ausgabe-Operationen ist, dass die Kommunikation potentiell schief gehen kann. Die Datei kann nicht lesbar sein, gar nicht existieren, oder das externe Betriebsmittel, in diesem Falle die Festplatte, es könnte aber auch eine Netzwerkverbindung sein, nicht mehr antworten. Um koordiniert auf Fehlerfälle reagieren zu können, kennt Java das Konzept der Ausnahmen, die auftreten können und mit einer Fehlerbehandlung abgefangen werden können. Dieses Konzept werden wir erst im nächsten Kapitel kennenlernen. So lange wir noch nicht wissen, wie wir diese Ausnahmefälle behandeln, können wir jede Methode, in der Ein-/Ausgabe-Operationen stattfinden mit `throws Exception` markieren. Damit geben wir an, dass es bei Ausführung dieser Methode zu Fehlern kommen kann.

```
4   public static String leseDatei(String dateiName)throws Exception{
5     String result = "";
```

Listing 6.27: TextLesen

Wir wollen aus einer Datei lesen. Somit erzeugen wir ein Objekt der Klasse `FileReader` und speichern dieses als einen `Reader`.

```
6     Reader read = new FileReader(dateiName);
```

Listing 6.28: TextLesen

Anders als man zunächst erwarten würde, hat die Klasse `Reader` keine Methode `read()`, die ein einzelnes `char` Zeichen zurück gibt, sondern eine Methode `read` mit dem Ergebnistyp `int`. Diese Zahl ist entweder die Unicode-Nummer des gelesenen Zeichens oder ein negativer Wert. Ist es ein negativer Wert, wird damit angezeigt, dass keine weiteres Zeichen mehr gelesen werden kann, dass wir am Ende des Zeichenstroms angelangt sind, weil in unserem Fall die Datei kein weiteres Zeichen mehr enthält. Deshalb sehen wir zunächst eine lokale Variable vom Typ `int` vor:

```
7     int i;
```

Listing 6.29: TextLesen

Nun können wir nacheinander die Zeichen aus der Datei lesen. Mindestens einmal müssen wir lesen, um zu schauen, ob es überhaupt Zeichen in der Datei gibt. Dann lesen wir so lange, bis das Ergebnis des Lesens eine negative Zahl ist. Hierzu bietet sich die `do-while`-Schleife an, die mindestens einmal durchlaufen wird.

```
8     do {
9       i = read.read();
```

Listing 6.30: TextLesen

Wenn wir ein Zeichen gelesen haben und dieses Zeichen eine Unicode-Nummer ist, können wir diese Zahl als `char`-Wert interpretieren:

```
10      if (i>=0){
11        char c = (char)i;
12        result = result+c;
13      }
14    } while (i>=0)
```

Listing 6.31: TextLesen

Wenn wir mit dem Lesen der Datei fertig sind, ist es sinnvoll, für das `Reader`-Objekt die Methode `close` aufzurufen, um dem Betriebssystem mitzuteilen, dass wir diese Ressource nicht weiter verwenden wollen.

```
15    read.close();
```

Listing 6.32: TextLesen

Somit haben wir eine sehr simple Methode, die den kompletten Inhalt einer Datei einliest.

```
16    return result;
17  }
18 }
```

Listing 6.33: TextLesen

Ganz analog geht das Schreiben von Textdateien, so dass sich mit beiden zusammen recht einfach ein kleines Programm zum Kopieren von Textdateien schreiben lässt:

```java
package name.panitz.oose.io;
import java.io.*;

public class Copy {
  public static void main(String[] args) throws Exception {
    Reader in = new FileReader(args[0]);
    Writer out = new FileWriter(args[1]);
    int c;
    while ((c = in.read()) >= 0){
      out.write(c);
    }
    out.close();
    in.close();
  }
}
```

Listing 6.34: Copy.java

6.3.2 Textcodierungen

`Reader` und `Writer` sind praktische Klassen zur Verarbeitung von Zeichenströmen. Primär sind aber auch Textdateien lediglich eine Folge von Bytes. Mit den Klassen `InputStreamReader` und `OutputStreamWriter` lassen sich Objekte vom Typ `IntputStream` bzw. `OutputStream` zu `Reader`- bzw. `Writer`-Objekten machen.

Statt die vorgefertigte Klasse `FileWriter` zum Schreiben einer Textdatei zu benutzen, erzeugt die folgende Version zum Kopieren von Dateien einen über einen `FileOutputStream` erzeugten `Writer` bzw. einen über einen `FileInputStream` erzeugten `Reader`:

```java
import java.io.*;

class Copy2 {
  static public void main(String [] args) throws Exception {

    Reader reader = new InputStreamReader(new FileInputStream(args[0])
    );
    Writer writer  = new OutputStreamWriter(new FileOutputStream(args
    [1]));

    int c;
    while ((c = reader.read()) >= 0){
      writer.write((char)c);
    }
    writer.close();
    reader.close();
  }
}
```

Listing 6.35: Copy2.java

Java-Strings sind Zeichenketten, die nicht auf eine Kultur mit einer bestimmten Schrift beschränkt, sondern in der Lage sind, alle im Unicode erfassten Zeichen darzustellen; seien es Zeichen der lateinischen, kyrillischen, arabischen, chinesischen oder sonst einer Schrift bis hin zur keltischen Keilschrift. Jedes Zeichen eines Strings kann potentiell eines dieser mehreren zigtausend Zeichen einer der vielen Schriften sein. In der Regel benutzt ein Dokument insbesondere im amerikanischen und europäischen Bereich nur wenige, kaum 100 unterschiedliche Zeichen. Auch ein arabisches Dokument wird mit weniger als 100 verschiedenen Zeichen auskommen.

Wenn ein Dokument im Computer auf der Festplatte gespeichert wird, so werden auf der Festplatte keine Zeichen einer Schrift, sondern Zahlen abgespeichert. Diese Zahlen sind traditionell Zahlen, die acht Bit im Speicher belegen, ein sogenanntes Byte. Ein Byte ist in der Lage, 256 unterschiedliche Zahlen darzustellen. Damit würde ein Byte ausreichen, alle Buchstaben eines normalen westlichen Dokuments in lateinischer Schrift (oder eines arabischen Dokuments) darzustellen. Für ein chinesisches Dokument reicht es nicht aus, die Zeichen durch ein Byte allein auszudrücken, denn es gibt mehr als 10000 verschiedene chinesische Zeichen. Es ist notwendig, zwei Byte im Speicher zu benutzen, um die vielen chinesischen Zeichen als Zahlen darzustellen.

Die *Zeichencodierung* (englisch: encoding) eines Dokuments gibt nun an, wie die Zahlen, die der Computer auf der Festplatte gespeichert hat, als Zeichen interpretiert werden sollen. Eine Codierung für arabische Texte wird den Zahlen von 0 bis 255 bestimmte arabische Buchstaben zuordnen, eine Codierung für deutsche Dokumente wird den Zahlen 0 bis 255 lateinische Buchstaben inklusive deutscher Umlaute und dem ß zuordnen. Für ein chinesisches Dokument wird eine *Codierung* benötigt, die den 65536 mit zwei Byte darstellbaren Zahlen jeweils chinesische Zeichen zuordnet. Man sieht, dass es *Codierungen* geben muss, die für ein Zeichen ein Byte im Speicher belegen, und solche, die zwei Byte im Speicher belegen. Es gibt darüberhinaus auch eine Reihe Mischformen; manche Zeichen werden durch ein Byte, andere durch zwei oder sogar durch drei Byte dargestellt.

Die Klasse `OutputStreamWriter` sieht einen Konstruktor vor, dem man zusätzlich zum `OutputStream`, in den geschrieben werden soll, als zweites Element auch die Codierung angeben kann, in der die Buchstaben abgespeichert werden sollen. Wenn diese Codierung nicht explizit angegeben wird, so benutzt Java die standardmäßig auf dem Betriebssystem benutzte Codierung.

In dieser Version der Kopierung einer Textdatei wird für den `Writer` ein Objekt der Klasse `OutputStreamWriter` benutzt, in der als Zeichenkodierung utf-16 benutzt wird.

```
1 import java.nio.charset.Charset;
```

```
2  import java.io.*;
3
4  class EncodedCopy {
5    static public void main(String [] args)throws Exception {
6      Reader reader = new FileReader(args[0]);
7      Writer writer = new OutputStreamWriter
8                           (new FileOutputStream(args[1])
9                           ,Charset.forName("UTF-16"));
10
11     int c;
12     while ((c = reader.read()) != -1){
13       writer.write(c);
14     }
15     writer.close();
16   }
17 }
```

Listing 6.36: EncodedCopy.java

Betrachtet man die Größe der geschriebenen Datei, so wird man feststellen, daß sie mehr als doppelt so groß ist wie die Ursprungsdatei.

```
sep@linux:~/fh/prog1/> java EncodedCopy EncodedCopy.java EncodedCopyUTF16.java
sep@linux:~/fh/prog1/> ls -l EncodedCopy.java
-rw-r--r--    1 sep      users         443 2004-01-07 19:12 EncodedCopy.java
sep@linux:~/fh/prog1/> ls -l EncodedCopyUTF16.java
-rw-r--r--    1 sep      users         888 2004-01-07 19:13 EncodedCopyUTF16.java
sep@linux:~/fh/prog1/>
```

Gängige Zeichencodierung sind:

- iso-8859-1: Damit lassen sich westeuropäische Texte mit den entsprechenden Sonderzeichen und Akzenten, wie in westeuropäischen Sprechen benötigt abspeichern. Jedes Zeichen wird mit einem Byte abgespeiechert. Andere Zeichen, sei es arabisch, chinesisch oder auch türkische Sonderzeichen, sind in dieser Zeichencodierung nicht abspeicherbar.
- utf-16: Hierbei hat jedes Zeichen genau zwei Byte in der Darstellung. Diese beiden Byte codieren exakt die Unicode-Nummer des Zeichens. Somit lassen sich in dieser Zeichencodierung alle Unicode-Zeichen abspeichern. Allerdings, wenn man nur lateinische Schrift in einem Text hat, hat eines dieser beiden Bytes immer den Wert 0. Es wird also viel Platz verschwendet.
- utf-8: Hier werden gängigen lateinische Zeichen als ein Byte codiert. Alle anderen Sonderzeichen oder Zeichen aus anderen Schriften werden mit mehreren Bytes codiert. Dieses hat den Vorteil, dass man alle Zeichen kodieren kann, für Texte in lateinischer Schrift aber nur ein Byte pro Zeichen benötigt. Es hat den Nachteil, dass unterschiedliche Zeichen unterschiedlich viel Platz auf der Festplatte benötigen. Man kann also einer Datei nicht ansehen, das wievielte Byte zum Beispiel das 1000. Zeichen des Textes ist.

6.3.3 Gepufferte Ströme

Die bisher betrachteten Ströme arbeiten immer exakt zeichenweise, bzw. byteweise. Damit wird bei jedem `read` und bei jedem `write` direkt von der Quelle bzw. an die Senke ein Zeichen übertragen. Für Dateien heißt das, es wird über das Betriebssystem auf die Datei auf der Festplatte zugegriffen. Handelt es sich bei Quelle/Senke um eine teure und aufwändige Netzwerkverbindung, so ist für jedes einzelne Zeichen über diese Netzwerkverbindung zu kommunizieren. Da in der Regel nicht nur einzelne Zeichen über einen Strom übertragen werden sollen, ist es effizienter, wenn technisch gleich eine Menge von Zeichen übertragen wird. Um dieses zu bewerkstelligen, bietet Java an, Ströme in gepufferte Ströme umzuwandeln.

Ein gepufferter Strom hat einen internen Speicher. Bei einer Datenübertragung wird für schreibende Ströme erst eine Anzahl von Zeichen in diesem Zwischenspeicher abgelegt, bis dieser seine Kapazität erreicht hat, um dann alle Zeichen aus dem Zwischenspeicher *en bloc* zu übertragen. Für lesende Ströme wird entsprechend für ein `read` gleich eine ganze Anzahl von Zeichen von der Datenquelle geholt und im Zwischenspeicher abgelegt. Weitere `read`-Operationen holen dann die Zeichen nicht mehr direkt aus der Datenquelle, sondern aus dem Zwischenspeicher, bis dieser komplett ausgelesen wurde und von der Datenquelle wieder zu füllen ist.

Die entsprechenden Klassen, die Ströme in gepufferte Ströme verpacken, heißen:

`BufferedInputStream, BufferedOutputStream` und

entsprechend `BufferedReader, BufferedWriter`.

Jetzt ergänzen wir zur Effizienzsteigerung noch das Kopierprogramm, so daß der benutzte `Writer` gepuffert ist:

```
1  import java.io.*;
2  import java.nio.charset.Charset;
3
4  class BufferedCopy{
5    static public void main(String [] args)throws Exception {
6      Reader reader = new BufferedReader(new FileReader(args[0]));
7      Writer writer = new BufferedWriter(new OutputStreamWriter
8                            (new FileOutputStream(args[1]),Charset.
     forName("UTF-16")));
9
10     int c;
11     while ((c = reader.read()) != -1){
12       writer.write(c);
13     }
14     writer.close();
15     reader.close();
16   }
17 }
```

Listing 6.37: BufferedCopy.java

6.3.4 Lesen von einem Webserver

Bisher haben wir nur aus Dateien gelesen und damit den Vorteil der Abstraktion der Ströme, dass die Quelle aus der gelesen wird recht unterschiedlich sein kann, noch nicht demonstriert. In diesem Abschnitt zeigen wir, wie man statt aus einer Datei von einem Webserver Dokumente lesen kann. Auch dabei werden Ströme verwendet. Damit gibt es ein einheitliches API zum Lesen von Information aus ganz unterschiedlichen Quellen. Einstiegspunkt zum Lesen von einem Webserver ist die Klasse `java.net.URL`, mit der die Adresse des Webservers angegeben werden kann. Die Klasse hat einen Konstruktor, der ein String-Argument erhält.

```
1  import java.io.InputStream;
2  import java.io.InputStreamReader;
3  import java.io.Reader;
4  import java.net.URL;
5  import java.net.URLConnection;
6
7  public class ReadFromServer {
8    public static void main(String[] args) throws Exception {
9
10     URL url = new URL("http://www.jarass.com/home/index.php");
```

Listing 6.38: ReadFromServer

Die Klasse `URL` hat eine Methode `openConnection`, die es erlaubt eine Verbindung über das Netzwerk aufzubauen.

```
11     URLConnection con = url.openConnection();
```

Listing 6.39: ReadFromServer

Von der damit erhaltenen Verbindung kann nun der Eingabestrom erfragt werden, also das Objekt, auf dem der Webserver uns Daten liefert.

```
12     InputStream in = con.getInputStream();
```

Listing 6.40: ReadFromServer

Von nun an geht alles, wie bereits bei dem Einlesen aus einer Datei:

```
13      Reader read = new InputStreamReader(in, "utf-8");
14      int i;
15      do {
16        i = read.read();
17        if (i>=0){
18          char c = (char)i;
19          System.out.print(c);
20        }
21      } while (i >= 0);
22        read.close();
23      }
24    }
25  }
```

Listing 6.41: ReadFromServer

6.3.5 Ströme für Objekte

Bisher haben wir uns darauf beschränkt, Zeichenketten über Ströme zu lesen und zu schreiben. Java bietet darüberhinaus die Möglichkeit an, beliebige Objekte über Ströme zu schreiben und zu lesen. Hierzu können mit den Klassen `ObjectOutputStream` und `ObjectInputStream` beliebige `OutputStream`- bzw. `InputStream`-Objekte zu Strömen für Objekte gemacht werden. In diesen Klassen stehen Methoden zum Lesen und Schreiben von Objekten zur Verfügung. Allerdings können über diese Ströme nur Objekte von Klassen geschickt werden, die die Schnittstelle `java.io.Serializable` implementieren. Die meisten Standardklassen implementieren diese Schnittstelle. `Serializable` enthält keine Methoden, es reicht also zum Implementieren aus, die Klausel `implements Serializable` für eine Klasse zu benutzen, damit Objekte der Klasse über Objektströme geschickt werden können.

Objektströme haben zusätzlich Methoden zum Lesen und Schreiben von primitiven Typen.

Folgendes Testprogramm schreibt eine Zahl und ein Listenobjekt in eine Datei, um diese anschließend wieder aus der Datei auszulesen.

```
1  import java.io.*;
2  import java.util.List;
3  import java.util.ArrayList;
4
5  public class WriteReadObject {
6    public static void main(String [] args) throws Exception{
```

Listing 6.42: WriteReadObject.java

Zunächst erzeugen wir einen Ausgabestrom:

```
7     OutputStream fos = new FileOutputStream("t.tmp");
```

Listing 6.43: WriteReadObject.java

Diesen verwenden wir nun, um eine in Strom, in den Objekte verschickt werden können, zu erzeugen.

```
8     ObjectOutputStream oos = new ObjectOutputStream(fos);
```

Listing 6.44: WriteReadObject.java

Wir brauchen irgendein beliebiges Objekt, dass wir nun speichern können. Hierzu legen wir beispielsweise eine Liste an:

```
9     List<String> xs = new ArrayList<String>();
10    xs.add("the");
11    xs.add("world");
12    xs.add("is");
13    xs.add("my");
14    xs.add("oyster");
```

Listing 6.45: WriteReadObject.java

Und nun können wir sowohl eine Zahl, aber auch das ganze Listenobjekt in die Datei speichern:

```
15    oos.writeInt(12345);
16    oos.writeObject(xs);
17    oos.close();
```

Listing 6.46: WriteReadObject.java

Nun können wir die geschriebene Datei wieder lesen und das Objekt dadurch neu erhalten. (realistischer Weise würde das natürlich in einem anderen Programm passieren). Hierzu brauchen wir einen EIngebaestrom:

```
18      FileInputStream fis = new FileInputStream("t.tmp");
```

Listing 6.47: WriteReadObject.java

Diesen machen wir zu einem Strom, aus dem Objekte gelesen werden können:

```
19    ObjectInputStream ois = new ObjectInputStream(fis);
```

Listing 6.48: WriteReadObject.java

Jetzt können wir daraus lesen. Zunächst die gespeicherte Zahl, dann das Listenobjekt.

```
20      int i = ois.readInt();
21      List<String> ys = (List<String>) ois.readObject();
22      ois.close();
23
24      System.out.println(i);
25      System.out.println(ys);
26    }
27  }
```

Listing 6.49: WriteReadObject.java

6.4 Ausnahmen

Es gibt während des Ablaufs eines Programms Situationen, die als Ausnahmen zum eigentlichen Programmablauf betrachtet werden können. Java hält ein Konzept bereit, das die Behandlung von Ausnahmen abseits der eigentlichen Programmlogik erlaubt.

6.4.1 Ausnahme- und Fehlerklassen

Java stellt Standardklassen zur Verfügung, deren Objekte einen bestimmten Ausnahme- oder Fehlerfall ausdrücken. Die gemeinsame Oberklasse aller Klassen, die Fehler- oder Ausnahmefälle ausdrücken, ist `java.lang.Throwable`. Diese Klasse hat zwei Unterklassen, nämlich:

- `java.lang.Error`: alle Objekte dieser Klasse drücken aus, dass ein ernsthafter Fehlerfall aufgetreten ist, der in der Regel von dem Programm selbst nicht zu beheben ist.
- `java.lang.Exception`: alle Objekte dieser Klasse stellen Ausnahmesituationen dar. Im Programm kann eventuell beschrieben sein, wie bei einer solchen Ausnahmesituation weiter zu verfahren ist. Eine Unterklasse von `Exception` ist die Klasse `java.lang.RuntimeException`.

6.4.2 Werfen von Ausnahmen

Ein Objekt vom Typ `Throwable` allein zeigt noch nicht an, dass ein Fehler aufgetreten ist. Hierzu gibt es einen speziellen Befehl, der im Programmablauf dieses kennzeichnet, der Befehl `throw`.

`throw` ist ein Schlüsselwort, dem ein Objekt des Typs `Throwable` folgt. Bei einem `throw`-Befehl verläßt Java die eigentliche Ausführungsreihenfolge des Programms und unterrichtet die virtuelle Maschine davon, dass eine Ausnahme aufgetreten ist.

Z.B. können wir für die Fakultätsmethoden bei einem Aufruf mit einer negativen Zahl eine Ausnahme werfen:

```
1  package name.panitz.exceptions;
2  public class FirstThrow {
3
4    public static int fakultät(int n){
5      if (n==0) return 1;
6      if (n<0) throw new RuntimeException();
7      return n*fakultät(n-1);
8    }
9
10   public static void main(String [] args){
11     System.out.println(fakultät(5));
12     System.out.println(fakultät(-3));
13     System.out.println(fakultät(4));
14   }
15 }
```

Listing 6.50: FirstThrow.java

Wenn wir dieses Programm starten, dann sehen wir, dass zunächst die Fakultät für die Zahl 5 korrekt berechnet und ausgegeben wird, dann der Fehlerfall auftritt, was dazu führt, dass der Fehler auf der Kommandozeile ausgegeben wird und das Programm sofort beendet wird. Die Berechnung der Fakultät von 4 wird nicht mehr durchgeführt. Es kommt zu folgender Ausgabe:

```
swe10:~> java name.panitz.exceptions.FirstThrow
120
Exception in thread "main" java.lang.RuntimeException
  at name.panitz.exceptions.FirstThrow.fakultät(FirstThrow.java:6)
  at name.panitz.exceptions.FirstThrow.main(FirstThrow.java:12)
swe10:~>
```

Wie man sieht, unterrichtet uns Java in der ersten Zeile davon, dass eine Ausnahme des Typs `RuntimeException` geworfen wurde. In der zweiten Zeile erfahren wir, dass dieses bei der Ausführung der Methode `fakultät` in Zeile 6 der Klasse `FirstThrow` geschehen ist. Anschließend, in den Zeilen weiter unten, gibt Java jeweils an, in welcher Methode der Aufruf der in der drüber liegenden Methode stattfand.

Die Ausgabe gibt also an, durch welchen verschachtelten Methodenaufruf es an die Stelle kam, in der die Ausnahme geworfen wurde. Diese Aufrufstruktur wird als Aufrufkeller (*stack trace*) bezeichnet.

Das Erzeugen eines Ausnahmeobjekts allein bedeutet noch keinen Fehlerfall. Wenn wir das obige Programm minimal ändern, so dass wir das Schlüsselwort `throw` weglassen, so wird der Sonderfall für negative Eingaben nicht gesondert behandelt.

```java
package name.panitz.exceptions;
public class NonThrow {

  public static int fakultät(int n){
    if (n==0) return 1;
    if (n<0) new RuntimeException();
    return n*fakultät(n-1);
  }

  public static void main(String [] args){
    System.out.println(fakultät(5));
    System.out.println(fakultät(-3));
    System.out.println(fakultät(4));
  }
}
```

Listing 6.51: NonThrow.java

Wenn wir dieses Programm starten, so wird es nicht terminieren und je nach benutzter Javamaschine schließlich abbrechen:

```
swe10:~> java name.panitz.exceptions.NonThrow
120

An irrecoverable stack overflow has occurred.
```

Es reicht also nicht aus, ein Fehlerobjekt zu erzeugen, sondern es muss dieses auch mit einem `throw`-Befehl geworfen werden. Geworfen werden können alle Objekte einer Unterklasse von `Throwable`. Versucht man hingegen, andere Objekte zu werfen, so führt dies schon zu einem Übersetzungsfehler.

Folgende Klasse:

```java
package name.panitz.exceptions;
public class NotThrowable {

  public static void main(String [] args){
      throw "i am not throwable";
  }
}
```

führt zu einem Übersetzungsfehler:

```
swe10:~> javac -d . NotThrowable.java
NotThrowable.java:5: incompatible types
found   : java.lang.String
```

```
required: java.lang.Throwable
      throw "i am not throwable";
            ^
1 error
swe10:~>
```

Ausnahmen können natürlich nicht nur auftreten, wenn wir sie selbst explizit geworfen haben, sondern auch von Methoden aus Klassen, die wir selbst benutzen, geworfen werden. So kann z.B. die Benutzung der Methode `charAt` aus der Klasse `String` dazu führen, dass eine Ausnahme geworfen wird.

```
1 package name.panitz.exceptions;
2 public class ThrowIndex {
3
4   public static void main(String [] args){
5       "i am too short".charAt(120);
6   }
7 }
```

Listing 6.52: ThrowIndex.java

Starten wir dieses Programm, so wird auch eine Ausnahme geworfen:

```
swe10:~> java name.panitz.exceptions.ThrowIndex
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
                                    String index out of range: 120
  at java.lang.String.charAt(String.java:516)
  at name.panitz.exceptions.ThrowIndex.main(ThrowIndex.java:5)
swe10:~>
```

Wie man an diesem Beispiel sieht, gibt Java nicht nur die Klasse der Ausnahme, die geworfen wurde, aus (`java.lang.StringIndexOutOfBoundsException:`), sondern auch noch eine zusätzliche Erklärung. Die Objekte der Unterklassen von `Throwable` haben in der Regel einen Konstruktor, der erlaubt noch eine zusätzliche Information, die den Fehler erklärt, mit anzugeben. Das können wir auch in unserem Beispielprogramm nutzen:

```
1  package name.panitz.exceptions;
2
3  public class SecondThrow {
4    public static int fakultät(int n){
5      if (n==0) return 1;
6      if (n<0)
7       throw
8        new RuntimeException
9             ("negative Zahl für Fakultätsberechnung");
10     return n*fakultät(n-1);
```

```
11    }
12
13    public static void main(String [] args){
14      System.out.println(fakultät(5));
15      System.out.println(fakultät(-3));
16      System.out.println(fakultät(4));
17    }
18  }
```

Listing 6.53: SecondThrow.java

Damit erhalten wir folgende Ausgabe:

```
swe10:~> java name.panitz.exceptions.SecondThrow
120
Exception in thread "main" java.lang.RuntimeException:
                                    negative Zahl für Fakultätsberechnung
  at name.panitz.exceptions.SecondThrow.fakultät(SecondThrow.java:6)
  at name.panitz.exceptions.SecondThrow.main(SecondThrow.java:12)
swe10:~>
```

Kapitel 7

Zusammenfassung und Ausblick

7.1 Fragen und Antworten

- **Ist Java eine kompilierte oder interpretierte Sprache?** Im Prinzip beides: Ein Compiler übersetzt den Quelltext in .class-Dateien, die Code für eine virtuelle Maschine enthalten. Ein Interpreter ist in der Lage diesen Code auszuführen.
- **Wie ist ein Javaprogramm strukturiert?** Pro Datei eine Klasse (oder Schnittstelle oder Aufzählung), die den Namen der Datei trägt.
- **Welche Namenskonventionen gibt es?** Klassennamen beginnen mit einem Großbuchstaben, globale Konstanten sind komplett in Großbuchstaben geschrieben. Pakete, Parameter, Variablen beginnen mit einem Kleinbuchstaben. Bei Bezeichnern aus mehreren Wörter fängt das nächste Wort immer mit einem Großbuchstaben an.
- **Ist Java statisch getypt?** Ja, der Compiler überprüft, ob das Programm korrekt getypt ist.
- **Gibt es auch dynamische Typüberprüfung zur Laufzeit?** Ja, zum einem verifiziert die virtuelle Maschine noch einmal den Code, zum anderen bewirkt eine Typzusicherung (cast) eine Laufzeittypüberprüfung.
- **Kann ich sicherstellen, daß die Typzusicherung während der Laufzeit nicht fehlschlägt.** Ja, indem vor der Typzusicherung der Test mit `instanceof` gemacht wird.
- **Wie schreibt man Unterprogramme oder Funktionen?** Immer innerhalb einer Klasse. Sie werden als Methoden bezeichnet.
- **Was sind die Eigenschaften einer Klasse?** Zum einen die Felder (auch als Attribute) bezeichnet, in denen Referenzen auf Objekte abgelegt werden können. Zum anderen die Methoden und Konstruktoren.
- **Was ist das `this`-Objekt?** Damit wird das Objekt bezeichnet, in dem sich eine Eigenschaft befindet.

- **Was bedeutet der `this`-Bezeichner, wenn ihm Parameter in Klammern folgen?** Dann bedeutet es den Aufruf eines weiteren Konstruktors aus derselben Klasse. Hierzu müssen Konstruktoren überladen sein. Dieser Aufruf kann nur als erster Befehl in einem Konstruktor stehen.
- **Was ist Vererbung?** Jede Klasse hat genau eine Oberklasse, die in der `extends`-Klausel angegeben wird. Objekte können alle Eigenschaften die in ihrer Oberklasse zur Verfügung stehen benutzen.
- **Was ist wenn ich keine Oberklasse angebe?** Dann ist automatisch die Klasse `Object` die Oberklasse.
- **Was sind Konstruktoren?** Konstruktoren sind der Code einer Klasse, der beim Erzeugen von neuen Objekten ausgeführt wird und in der Regel die Felder des Objektes initialisiert. Konstruktoren wir mit dem `new`-Befehl beim Erzeugen von Objekten aufgerufen.
- **Wie werden Konstruktoren definiert?** Ähnlich wie Methoden. Sie haben den Namen der Klasse und keinen Rückgabetyp.
- **Wie werden Konstuktoren aufgerufen?** Durch das Schlüsselwort `new` gefolgt von dem Klassennamen.
- **Hat jede Klasse einen Konstruktor?** Ja.
- **Und wenn ich keinen Konstruktor für meine Klasse schreibe?** Dann fügt Java einen leeren Konstruktor ohne Parameter ein.
- **Was ist der Unterschied zwischen statischen und nicht statischen Eigenschaften einer Klasse?** Eine statische Eigenschaft ist nicht an spezielle Objekte gebunden. Sie hat daher auch kein `this`-Objekt. Eine statische Eigenschaft existiert nur einmal für alle Objekte einer Klasse. Nichtstatische Methoden werden auch als Objektmethoden bezeichnet.
- **Gibt es auch statische Konstruktoren?** Ja, pro Klasse genau einen, der keine Parameter hat.
- **Kann ich den statischen Konstruktor auch selbst definieren?** Ja, mit dem Schlüssenwort `static` gefolgt von in geschweiften Klammern eingeschlossenen Code.
- **Was bedeutet Überschreiben von Methoden?** Methoden, die es in der Oberklasse bereits gibt in einer Unterklasse neu zu definieren.
- **Kann ich auch Konstruktoren überschreiben?** Nein.
- **Was bezeichnet man als Polymorphie?** Wenn eine Methode in verschiedenen Unterklassen einer Klasse überschrieben wird.
- **Kann ich in einer überschreibenen Methode, die überschriebene Methode aufrufen?** Ja, indem man das Schlüsselwort `super` benutzt und mit einem Punkt abgetrennt den eigentlichen Methodenaufruf folgen lässt.

- **Kann ich Konstruktoren der Oberklasse aufrufen.** Ja, aber nur im Konstruktor als erste Anweisung. Hier muss sogar der Aufruf eines Konstruktors der Oberklasse stehen. Dieser Aufruf wird durch das Schlüsselwort `super` gefolgt von der Parameterliste gemacht.
- **Was ist, wenn ich im Konstruktor keinen Aufruf an einen Konstruktor der Oberklasse schreiben.** Dann generiert Java den Aufruf eines Konstruktors der Oberklasse ohne Parameter als erste Anweisung in den Konstruktor. Sollte so ein parameterloser Konstruktor nicht existieren, dann gibt es allerdings einen Folgefehler.
- **Was ist späte Bindung (late binding)?** Beim Aufruf von Objektmethoden wird immer der Methodencode ausgeführt, der in der Klasse implementiert wurde, von der das Objekt, auf dem diese Methode aufgerufen wurde, erzeugt wurde. Es wird also immer die überschreibende Version einer Methode benutzt.
- **Funktioniert späte Bindung auch für Felder?** Nein.
- **Funktioniert späte Bindung auch für statische Methoden?** Nein.
- **Funktioniert späte Bindung auch für Konstruktoren?** Nein.
- **Was sind überladene Methoden?** Methoden gleichen Namens in einer Klasse, die sich in Typ/Anzahl der Parameter unterscheiden.
- **Können auch Konstruktoren überladen werden?** Ja.
- **Gibt es das Prinzip von später Bindung auch für die verschiedenen überladenen Versionen einer Methode?** Nein! Die Auflösung, welche der überladenen Versionen einer Methode ausgeführt wird, wird bereits statisch vom Compiler vorgenommen und nicht dynamisch während der Laufzeit.
- **Was sind abstrakte Klassen?** Klassen, die als `abstract` deklariert sind. Nur abstrakte Klassen können abstrakte Eigenschaften enthalten.
- **Und was sind abstrakte Eigenschaften?** Das sind Methoden, die keinen Methodenrumpf haben.
- **Können abstrakte Klassen Konstruktoren haben?** Ja, allerdings können von abstrakten Klassen keine Objekte mit `new` erzeugt werden.
- **Wie kann ich dann Objekte einer abstrakten Klassen erzeugen und wozu haben die dann Konstruktoren?** Indem Objekte einer nicht abstrakten Unterklasse mit `new` erzeugt werden. Im Konstruktor der Unterklasse wird ein Konstruktor der abstrakten Oberklasse aufgerufen.
- **Kann eine Klasse mehrere abstrakte Oberklassen haben?** Nein, auch abstrakte Klassen sind Klassen und es gilt die Regel: jede Klasse hat genau eine Oberklasse.
- **Kann ich in einer abstrakten Klasse abstrakte Methoden aufrufen?** Ja, im Rumpf einer nicht-abstrakten Methode können bereits abstrakte Klassen aufgerufen werden.

- **Was sind Schnittstellen?** Man kann Schnittstellen als abstrakte Klassen ansehen, in denen jeder Methode abstrakt ist.
- **Warum gibt es dann Schnittstellen?** Schnittstellen gelten nicht als Klassen. Eine Klasse kann nur eine Oberklasse haben, aber zusätzlich mehrere Schnittstellen implementieren.
- **Wie wird deklariert, daß eine Klasse eine Schnitstelle implementiert?** Durch die `implements`-Klausel in der Klassendeklaration. In ihr können mehrere Komma getrennte Schnittstellen angegeben werden.
- **Kann eine Schnittstelle eine Oberklasse haben?** Nein.
- **Kann eine Schnittstelle weitere Oberschnittstellen haben.** Ja, diese werden in der `extends`-Klausel angegeben.
- **Muss eine Klasse die eine Schnittstelle implementiert alle Methoden der Schnittstelle implementieren?** Im Prinzip ja, jedoch nicht, wenn die Klasse selbst abstrakt ist.
- **Haben Schnittstellen auch Objektfelder?** Nein! Nur Methoden und statische Felder.
- **Sind alle Daten in Java Objekte?** Nein, es gibt 8 primitive Typen, deren Daten keine Objekte sind. Es gibt aber zu jeden dieser 8 primitiven Typen eine Klasse, die die Daten entsprechend als Objekt speichern.
- **Welches sind die primitiven Typen?** byte, short, int, long, double, float, char, boolean
- **Sind Strings Objekte?** Ja, die Klasse String ist eine Klasse wie Du und ich.
- **Sind Reihungen (arrays) Objekte?** Ja, und sie haben sogar ein Attribut, das ihre Länge speichert.
- **Was sind generische Typen?** Klassen oder Schnittstellen, in denen ein oder mehrere Typen variabel gehalten sind.
- **Wie erzeugt man Objekte einer generischen Klasse?** Indem beim Konstruktoraufruf in spitzen Klammern konkrete Typen für die Typvariablen angegeben werden.
- **Was passiert, wenn ich für generische Typen die spitzen Klammern bei der Benutzung weglasse?** Dann gibt der Compiler eine Warnung und nimmt den allgemeinsten Typ für die Typvariablen an. Meistens ist das der Typ `Object`.
- **Was sind generische Methoden?** Methoden in denen ein oder mehrere Parametertypen variabel gehalten sind.
- **Was sind typische Beispiel für generische Typen?** Alle Sammlungsklassen und Abbildungsklassen im Paket `java.util`, zB die Klassen `ArrayList`,`LinkedList`, `HashSet`, `Vector` oder die Schnittstellen `List`, `Map`, `Set`.

- **Wo wir gerade dabei sind. Was sollte ich bei der Klasse `Vector` beachten?** In 90% der Fälle ist ein Objekt der Klasse `ArrayList` einem Objekt der Klasse `Vector` vorzuziehen. `Vector` ist eine sehr alte Klasse. Ihre Objekte sind synchronisiert, die anderen Sammlungsklassen nicht, es lassen sich von den anderen Sammlungsklassen allerdings synchronisierte Kopien machen. Am besten `Vector` gar nie benutzen.
- **Was ist automatisches Boxing und Unboxing?** Die Konvertierung von Daten primitiver Typen in Objekte der korrespondierenden Klassen und umgekehrt wird automatisch vorgenommen.
- **Welche zusammengesetzten Befehle gibt es?** if, while, for, switch
- **Was hat es mit der besonderen for-Schleife auf sich?** Es handelt sich um eine sogenannte for-each-Schleife. Syntaktisch trennt hier ein Doppelpunkt die lokale Schleifenvariable das Sammlungsobjekt.
- **Für welche Objekte kann die for-each-Schleife benutzt werden?** Für alle Objekte, die die Schnittstelle `Iterable` implementieren und für Reihungen.
- **Bedeutet das, ich kann die for-each Schleife auch für meine Klassen benutzen?** Ja, genau, man muß nur die Schnittstelle `Iterable` hierzu implementieren.
- **Können Operatoren überladen oder neu definiert werden?** Nein.
- **Was sind Ausnahme?** Objekte von Unterklassen der Klasse `Exception`. Zusätzlich gibt es die Klasse `Error` und die gemeinsame Klasse `Throwable`.
- **Wozu sind Ausnahmen gut?** Um in bestimmten Situationen mit einem `throw` Befehl den Programmfluß abzubrechen um eine besondere Ausnahmesituation zu signalisieren.
- **Was für Objekte dürfen in eine `throw` geworfen werden?** Nur Objekte einer Unterklasse von `Throwable`.
- **Wie werden geworfene Ausnahme behandelt?** Indem man sie innerhalb eines `try-catch`-Konstruktes wieder abfängt.
- **Kann man auf unterschiedliche Ausnahmen in einem `catch` unterschiedlich reagieren?** Ja, einfach mehrere `catch` untereinanderschreiben. Das als erstes zutreffende `catch` ist dann aktiv.
- **Was ist das mit dem `finally` beim `try-catch`?** Hier kann Code angegeben werden, der immer ausgeführt werden soll, egal ob und welche Ausnahme aufgetreten ist. Dieses ist sinnvoll um eventuell externe Verbindungen und ähnliches immer sauber zu schließen.
- **Darf man beliebig Ausnahmen werfen?** Ausnahmen, die nicht in einem `catch` abgefangen werden müssen in der `throws`-Klausel einer Methode deklariert werden.

- **Alle?** Nein, Ausnahmeobjekte von `RuntimeException` dürfen auch geworfen werden, wenn sie nicht in der `throws`-Klausel einer Methode stehen.
- **Wozu sind Pakete da?** Unter anderen damit man nicht Milliarden von Klassen in einem Ordner hat und dabei die Übersicht verliert.
- **Ist mir egal. Pakete sind mir zu umständlich.** Das ist aber dumm, wenn man Klassen gleichen Namens aus verschiedenen Bibliotheken benutzen will. Die kann man nur unterscheiden, wenn sie in verschiedenen Paketen sind.
- **OK, ich sehs ein. Wie nenne ich mein Paket?** Am besten die eigene Webdomain rückwärsts nehmen also für uns an der FH: `de.hsrm.informatik.panitz.meineApplikation`
- **Da fehlt doch der Bindestrich.** Der ist kein gültiges Zeichen innerhalb eines Bezeichners in Java (es ist ja der Minusoperator). Bindestriche in Webadressen sind eine recht deutsche Krankheit.
- **Muß ich imports machen um Klassen aus anderen Paketen zu benutzen?** Nein. Das macht die Sache nur bequemer, weil sonst der Klassenname immer und überall komplett mit seinem Paket angegeben werden muß, auch beim Konstruktoraufruf oder z.B. `instanceof`.
- **Machen Import-Anweisungen das Programm langsamer oder größer?** Nein! Sie haben insbesondere nichts mit includes in C gemein. Sie entsprechen eher dem `using namespace` aus C++.
- **Da gibt es doch auch noch `public`, `private` und `protected`.** Jaja, die Sichtbarkeiten. Hinzu kommt, wenn man keine Sichtbarkeit hinschreibt. `public` heißt von überall aufrufbar, `protected` heißt in Unterklassen und gleichem Paket aufrufbar, `package` das ist wenn man nichts hinschreibt, heißt nur im gleichen Paket sichtbar, und `private` nur in der Klasse.
- **Und dann war da noch das `final`.** Das hat zwei Bedeutungen: bei Variablen und Feldern, daß sie nur einmal einen Wert zugewiesen bekommen, bei Klassen, daß keine Unterklassen von der Klasse definiert werden
- **Ich will GUIs programmieren.** Wunderbar, da gibt es eine ganze Reihe Bibliotheken, unter andere n auch Swing und AWT.
- **Na toll, warum zwei?** Historisch.
- **Und welche sol ich jetzt benutzen?** Swing! (oder sich für die Zukunft JavaFX anschauen)
- **Also kann ich alles über AWT vergessen, insbesondere das Paket `java.awt`?** Nein. Swing benutzt große Teile aus AWT. Insbesondere hat jede Swingkomponente eine Oberklasse aus AWT. Aber auch die Ereignisbehandlungsklassen werden vollständig aus AWT benutzt.
- **Und woran erkenne ich jetzt, wann ich Komponente aus AWT benutzen muss, obwohl ich in Swing programmiere?** Alle graphischen

AWT-Komponenten haben ein Swing Pendant, das mit dem dem Buchstaben `J` beginnt, z.B. `javax.swing.JButton` im Gegensatz zu `java.awt.Button`.

- **Sonst noch paar Tipps, zum benutzen von Swing?** Keine eigenen Threads schreiben! Nicht die Methode `paint` sondern `paintComponent` überschreiben.
- **Toll, wenn ich keine Threads schreiben soll, wie mache ich dann z.B. zeitgesteuerte Ereignisse in Swing?** Da gibt es die Klasse `javax.swing.Timer` für.
- **Na ich weiß nicht, ob ich jetzt genügend weiß für die Klausur?** Das weiß ich auch nicht, über alles haben wir hier nicht gesprochen. Ich denke aber es reicht an Wissen aus.

Aufgabe 1 (Modellieren und Schreiben von Klassen)

Lösen Sie diese Aufgabe auf dem Papier.

Sie sollen in dieser Aufgabe Klassen entwickeln, die es erlauben, Informationen über die Personalstruktur einer Firma zu speichern.

a) Entwickeln Sie eine Klasse `Mitarbeiter`. Ein Mitarbeiter-Objekt dieser Klasse repräsentiert einen Angestellten in einer Firma. Es soll die folgenden Eigenschaften haben: der Name des Mitarbeiters, der durch eine Zeichenkette dargestellt ist, eine ganze Zahl, die das Monatsgehalt des Mitarbeiters in Cent ausdrückt und eine ganze Zahl, die die wöchentliche Arbeitszeit des Mitarbeiters darstellt.

   Schreiben Sie einen Konstruktor und überschreiben Sie die Methode `equals` auf adequate Weise. Überschreiben Sie auch die Methode `toString` auf sinnvolle Weise.

b) Schreiben Sie eine Unterklasse `Teamleiter` der Klasse `Mitarbeiter`. `Teamleiter`-Objekte repräsentieren Mitarbeiter, die ein Team leiten. Diese enthalten zusätzlich ein Feld einer Standardliste von Mitarbeiterobjekten, die das Team darstellen, und ein String-Feld für den Teamnamen.

   Die Methode `toString` soll überschrieben werden, um die zusätzliche Information mit zu berücksichtigen.

c) Schreiben Sie in der Klasse `Teamleiter` eine Methode:

```
public int durchschnittsgehalt()
```

   Sie soll das Durchschnittsgehalt der Teammitglieder des Teams, das der Teamleiter leitet berechnen.

Aufgabe 2 a) Führen Sie die folgende Klasse von Hand aus. Schreiben Sie dabei auf, in welcher Reihenfolge die Zeilen durchlaufen werden und mit welchen Werten die einzelnen Variablen während des Programmdurchlaufs belegt sind. Schreiben Sie auf, was auf dem Bildschirm ausgegeben wird.

```
1  class Aufgabe2a{
2    public static void main(String [] args){
3      int y = 9;
4      int x = 39;
5      while (x > 20){
6        System.out.println(x+" "+y);
7        switch (x%3){
8            case 2: x = x-1;
9                    y /= 2;
10           case 1: x = x-1;
11                   y = y*10;
12                   break;
13           default: x = x-1;
14       }
15       if (x<27) break;
16     }
17   }
18 }
```

Listing 7.1: Aufgabe2a.java

b) Betrachten Sie folgende Klasse:

```
1
2  class Aufgabe2b{
3    static int f(int x,int y1,int y2){
4       System.out.println("x="+x+" y1="+y1+" y2="+y2);
5       return g(y1*y1,x+2,y2+3);
6    }
7    static int g(int x1,int x2,int y){
8       System.out.println("x1="+x1+" x2="+x2+" y="+y);
9      if (0==y) return x1;
10     return f(x1,x2-1,y-4);
11   }
12   static int g(int x,int y){
13     return g(x,x,y);
14   }
15   public static void main(String [] args){
16    System.out.println(g(2,4));
17   }
18 }
```

Listing 7.2: Aufgabe2b.java

Berechnen Sie schrittweise das Ergebnis des Ausdrucks `g(2,4)`.

Aufgabe 3 (SwingGUI)

Lösen Sie auch diese Aufgabe auf einem Blatt Papier.

Gegeben Sei folgende Schnittstelle:

```
1 interface Function{
2   int rechne(int x,int y);
3 }
```

Listing 7.3: Logik.java

a) Schreiben Sie eine Klasse `Rechner`, die von der Klasse `JPanel` erbt. Sie enthalte fünf Felder: ein Objekt des Typs `Function`, ein `JButton`-Objekt, zwei `JTextField`-Objekte und ein `JLabel`-Objekt. Die `JTextField`-Objekte sind zwei für Eingaben gedacht, und das `JLabel`-Objekt für die Ausgabe.

   Implementieren Sie die Klasse so, dass bei Drücken des Knopfes der Text der zwei Textfelder zur Eingabe gelesen wird (`getText()`) und die Eingabetexte in ganze Zahlen umgewandelt werden. Diese werden dann der Methode `rechne` des `Function`-Objekts übergeben. Das Ergebnis dieses Methodenaufrufs soll in das JLabel als Text geschrieben werden (`setText(String s)`).

   Der Kontruktor der Klasse soll genau einen Parameter des Typs `Function` haben.

b) Schreiben Sie eine Applikation mit einer `main`-Methode, in der ein Fenster, mit einem `Rechner`-Objekt geöffnet wird. Bei Drücken des Knopfes sollen die zwei eingegebenen Zahlen addiert werden und im Ausgabefeld erscheinen.